AFRL-IF-RS-TR-2003-244
Final Technical Report
October 2003

FLEXIBLE DECISION SUPPORT IN DEVICE-SATURATED ENVIRONMENTS

Cornell University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. H575

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2003-244 has been reviewed and is approved for publication.

APPROVED: /s/
BRADLEY J. HARNISH
Project Engineer

FOR THE DIRECTOR: /s/
WARREN H. DEBANY, JR., Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	OCTOBER 2003	Final Apr 99 – Mar 03

4. TITLE AND SUBTITLE
FLEXIBLE DECISION SUPPORT IN DEVICE-SATURATED ENVIRONMENTS

5. FUNDING NUMBERS
C - F30602-99-2-0528
PE - 62301E
PR - H575
TA - 16
WU - 01

6. AUTHOR(S)
Johannes Gehrke

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Cornell University
4105B Upson Hall
Ithaca New York 14853

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington Virginia 22203-1714

AFRL/IFGA
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2003-244

11. SUPPLEMENTARY NOTES
AFRL Project Engineer: Bradley J. Harnish/IFGA/(315) 330-1884/ Bradley.Harnish@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***
The widespread distribution of small-scale sensors, actuators, and embedded processors is transforming the physical world into a computing platform. Sensor networks with nodes that combine physical sensing capabilities with networking and computation capabilities have become ubiquitous. Existing sensor networks assume that the sensors are preprogrammed and send data to a central frontend where the data is processed. This approach has two major drawbacks. First, the user cannot change the behavior of the system on the fly. Second, communication in today's networks is orders of magnitude more expensive than local computation, thus in-network processing can vastly reduce resource usage. We investigated a database approach to unite the requirements of scalability and flexibility in monitoring the physical world. We built a new distributed data management infrastructure that scales with the growth of sensor interconnectivity and computational power on the sensors over the next decades. Our system resides directly on the sensor nodes and creates the abstraction of a single processing node without centralizing data or computation. Our system provides scalable, fault-tolerant, flexible data access and intelligent data reduction, and its design involved a confluence of novel research in database query processing, data mining, networking, and distributed systems.

14. SUBJECT TERMS
Sensor Network, Data Management, Query Processing, Monitoring The Physical World

15. NUMBER OF PAGES
83

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

List of Figures

1 Introduction

The objective of this effort was to develop a scalable declarative programming environment that facilitates flexible decision-making in sensor networks.

We believe that procedural programming paradigms are inappropriate for highly scalable and flexible applications built on sensor networks. Realistic applications should instead be developed declaratively based on a high-level logical abstraction of the system. This approach separates the behavior of the application (specified by the queries) from its implementation, leading to the promise of efficiency, flexibility, and scalability.

In the Cougar project at Cornell University, we have developed distributed database techniques to process queries over sensor networks. Our approach allows users to formulate queries that involve functionalities provided by sensors out in the network. Our system then optimizes these queries to (a) reduce response latency and to (b) take maximal advantage of the resources available in the sensor network while at the same time balancing the utilization of critical resources.

The Cougar system resides directly on the sensor nodes and creates the abstraction of a single processing node without centralizing data or computation. Cougar provides scalable, fault-tolerant, flexible data access and intelligent data reduction. Its design involves a confluence of novel research in database query processing, data mining, networking, and distributed systems.

2 Reporting Period: This is the final annual report.

3 System Overview

Cougar has a three-tier architecture:

- *QueryProxy* - a small database component that runs on sensor nodes to interpret and execute queries.
- *Frontend* - a more powerful *QueryProxy* that serves as a gateway for connections to the world outside of the sensor network.
- *GUI* - a graphical user interface through which users can pose ad-hoc and long-running queries on the sensor network.

Our system forms clusters out of the sensors to allow intelligent in-network aggregation to conserve energy by reducing the amount of communication between sensor nodes. The query processing component handles distributing queries to the sensor nodes and retrieving and processing results before sending the data to the user.

The *QueryProxy* runs on each sensor node in the network, while the *FrontEnd* runs on selected nodes and acts as a gateway between the *GUI* and the *QueryProxies*. The *QueryProxy* and *FrontEnd* are built using C++, and the GUI using Java.

The *QueryProxy* and *FrontEnd* run under Linux on either x86 or Sensoria WINS NG node hardware. The *GUI* runs on any platform that supports Java and Swing. Communications within the sensor network are transmitted using ISI's Directed Diffusion and are formatted as XML. The *GUI* and *FrontEnd* communicate over TCP/IP sockets.

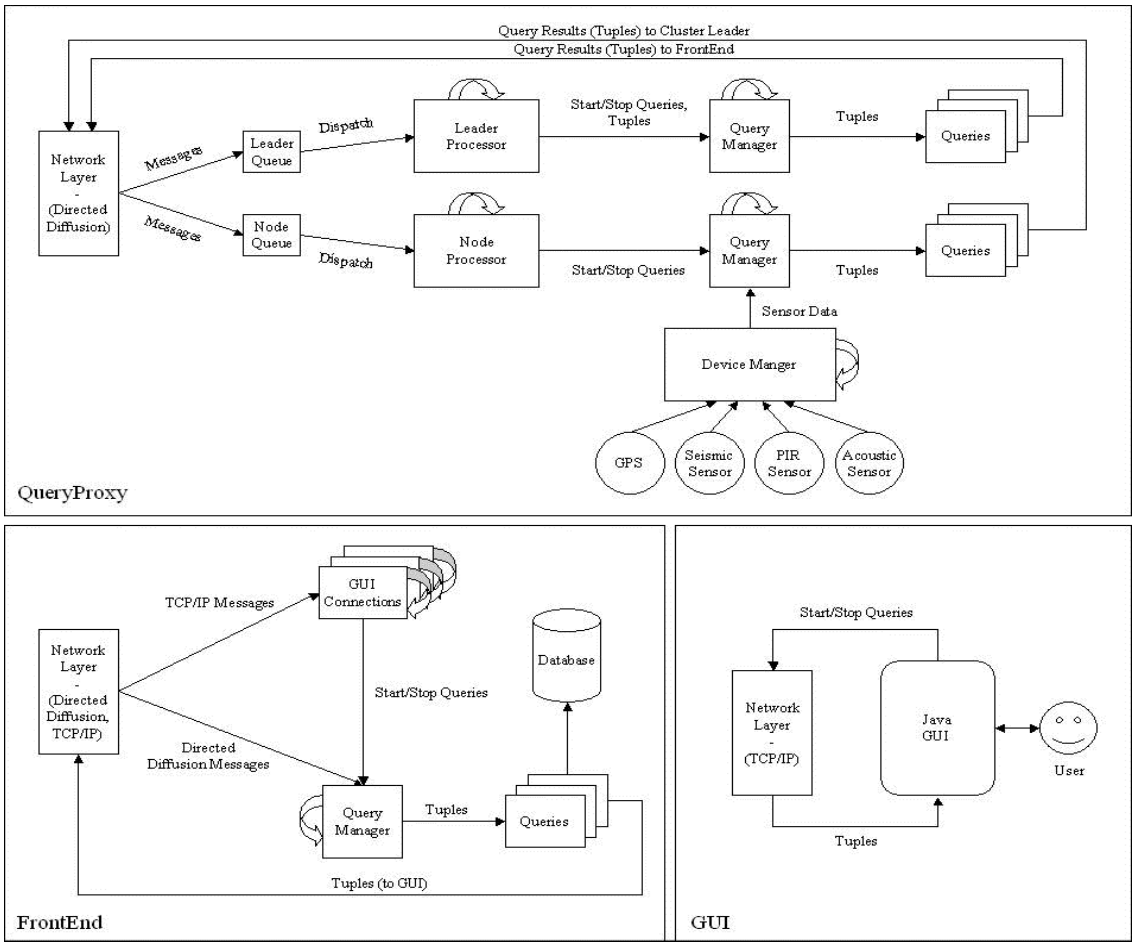

Figure 1: System Overview

3.1 QueryProxy

The *QueryProxy* consists of three parts: the device manager, the node layer, and the leader layer. The sensor nodes are capable of acting as leaders or normal query processing/signal processing nodes. When the network is set up, clusters are formed and leaders are elected from the nodes in the clusters. The *QueryProxy* system has a hierarchical structure, with the FrontEnd communicating with nodes that act as cluster leaders, and with cluster leaders communicating with the FrontEnd and with the other sensor nodes in their clusters.

The device manager takes readings from the sensors. On the Sensoria nodes this involves interacting with the BAE Systems signal processing module. BAE's signal processing can generate either raw readings directly from the sensors or high level events (e.g. detections). The device manager is fully integrated for processing all of these types of data.

The node layer manages the execution of queries on the sensor node and the interaction with the device manager. This code is active on all the nodes. When a query is being processed, the node layer requests the required tuples from the device manager. Then, the query is processed using those tuples, and the results are sent to the cluster leader.

The cluster leader is a specially designated node in the cluster that is elected when the system forms clusters. In addition to its node processing layer it has an active leader processing layer which receives tuples from the other members of the cluster. The leader layer then processes the queries using the received tuples and sends the replies to the FrontEnd that initiated the query. When appropriate, tuples are aggregated before being sent out.

3.2 FrontEnd

The *FrontEnd* issues queries it has received from the *GUI* to the *QueryProxy* software running on the sensors. It keeps track of the queries currently running for the *GUIs* running on the system and receives messages from nodes that are cluster leaders. The *FrontEnd* delivers each tuple to the interested queries, does some processing of the tuples, and sends a response to the *GUI* that initiated the query. It can also output tuples to a remote MySQL or Postgres database.

3.3 GUI

The GUI allows the user to pose queries using SQL and to display query results in tabular format. A map component allows the user to visualize the topology of sensors in the network. The user can collect nodes into clusters which automatically elect a leader to communicate with the FrontEnd. The map is also used to specify a region that the query

should run over (Figure 2). Results from the query are displayed in a simple table for easy viewing (Figure 3). Event queries such as detection and tracking queries are displayed visually on the map in real time (see Figures 4-5).

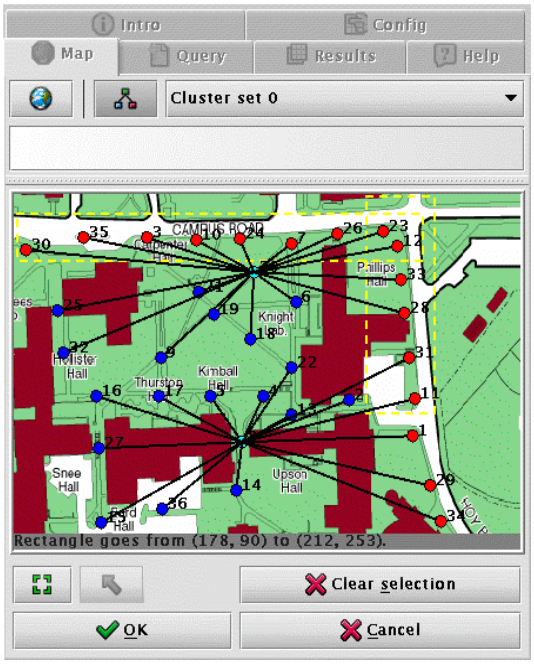

Figure 2: GUI Map Selection

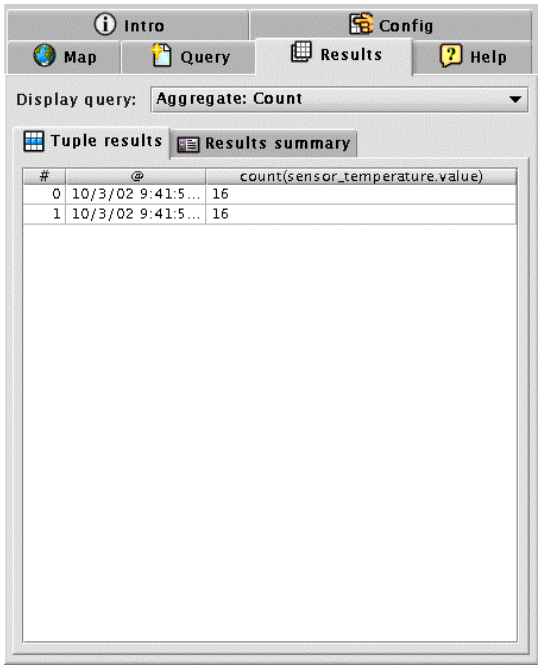

Figure 3: Query Results

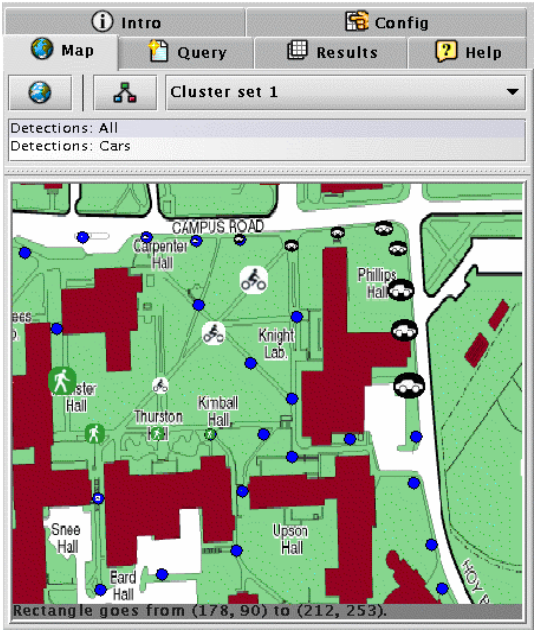

Figure 4: Detection Results

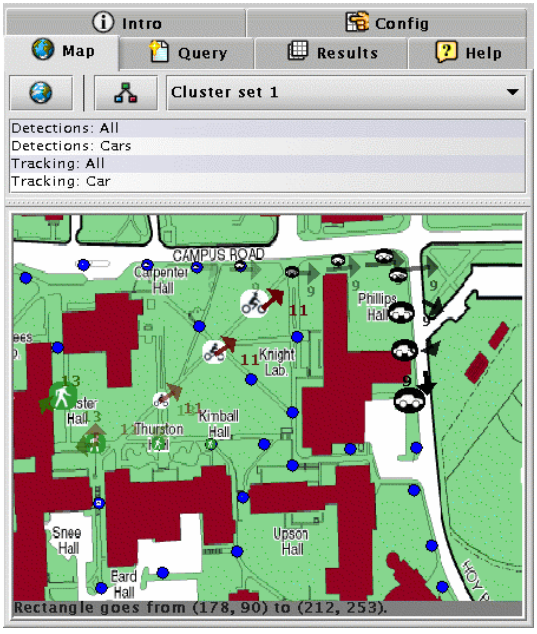

Figure 5: Tracking Results

4 System Internals

4.1 Directed Diffusion Interests

The *QueryProxy* component and the *FrontEnds* communicate with each other using ISI's directed diffusion. Four types of interests are set up:

1. Leader -> Front End channel. The front ends in the network all subscribe to a common Interest. This allows all Leaders in the network to publish data for the Front Ends to receive. The Leader nodes use this channel to send query results back to the front end that initiated the query.
2. Front End -> All Nodes channel. All nodes in the network subscribe to a common Interest. This allows all Front Ends in the network to publish data for all nodes on the network to receive. The Front Ends use this channel to send queries to leaders and to create clusters.
3. Cluster Nodes -> Cluster Leader channel. Each cluster leader subscribes to an Interest that allows it to receive published query results from other nodes in its cluster. Each cluster will have a different CN->CL channel.
4. Cluster Leader -> Cluster Nodes channel. Each node in a cluster subscribes to an Interest common to that cluster. This allows the cluster leader to publish queries for the cluster nodes to process. This channel is also used for leader selection.

4.2 Creating Clusters

The user at a FrontEnd creates a cluster by specifying the number of clusters to create in the GUI. The GUI will then proceed and choose a subset of the nodes in the network to act as centeroids. The centeroids act as cluster leaders. Then, the centeroids are broadcast to all nodes in the network. Nodes that are centeroids act as cluster leaders, and other nodes join the cluster that has the closest centeroid.

Each node can belong to multiple clusters. The user can specify a box region and nodes within that region are grouped into a single cluster. With this functionality, the user can optimize the clustering for better performance according the query workload and distribution of sensor nodes.

4.3 Query Types

Several different types of queries are supported by the Cougar system:

1. Network Status Queries – What is the current operational status of nodes in region *R*?

2. Periodic Sensor Queries - Return the light reading of each sensor in region *R* every *S* seconds.
3. Aggregated Sensor Queries - Calculate the average temperature of the sensors in region *R* every *S* seconds.
4. Detection Queries - Report the detection of any vehicle in region *R*.
5. Perimeter Queries - Alert whenever an intruder enters region *R*.
6. Trigger Queries – Trigger alarm when the average temperature goes above a threshold *T* in region *R*.
7. Tracking Queries – Show tracks for vehicles heading North in region *R*.

4.4 Query Processing

When a query is posed at the FrontEnd, the FrontEnd translates the query into a XML message that the leader nodes understand. This message is sent to all leaders within the network. The leader node is responsible for checking to see if the query involves any nodes within its cluster region. If it does, then the leader must generate a plan to execute the query. After the plan is generated, the leader node sends out tasks for the nodes within its cluster to execute. The leader node is then responsible for receiving replies from the nodes within its cluster and generating an appropriate reply for the FrontEnd. The FrontEnd may need to combine results received from various leader nodes before presenting it to the user who posed the query, as each leader node is only responsible for its own cluster region, and a query may involve multiple clusters.

4.5 Steps to Pose a Query

1. FrontEnd receives a query in SQL form from the user. In addition, the user must also specify a region in which the query applies, as well as a duration and frequency.
2. FrontEnd translates the SQL query and duration and frequency into a XML Query message and sends it to the cluster leaders.
3. All nodes that belong to clusters that have regions that intersect the region being queried will process the message.
4. Leader nodes will generate a plan for the execution of the query. Then, the leader nodes will execute the query by publishing XML Query messages to its cluster nodes.
5. The cluster nodes will place the query in their query caches. This is necessary in case the leader node fails. If this occurs, another node must take its place as the leader and will need to know what queries are currently being executed.
6. Periodically, as specified by the Frequency supplied by the user, the cluster nodes will send back results to the leader.
7. The Cluster leader will receive XML Responses from its cluster nodes periodically. It must process these responses appropriately and, at the user supplied frequency, must return a result to the FrontEnd.

8. The FrontEnd receives replies from cluster leaders and processes their results accordingly and updates query results at the supplied frequency.
9. Repeat 5 through 7 until the query is complete.

A node may also receive a delete query message which is identical to the create query message except for a tag. The node will then remove the query from its internal structures so that it no longer schedules the query to be executed and no longer sends results back.

Certain queries such as aggregation queries are modified by the leader nodes before being send to the other nodes in the cluster. The leader node in this case would remove the aggregation from the query before sending it out to the nodes. When it receives all the results, it performs the aggregation and returns the value to the FrontEnd.

5 Source Level Documentation

5.1 Code Layout and Directory Structure

- `queryproxy`: Contains makefiles for the components of the system
  - `bae`: Code for running BAE's signal processing module
  - `FrontEnd2`: Code for the FrontEnd. Pulls in some code from the QueryProxy2 directory.
  - `GUI`: Java GUI code for querying the sensor network
    - `Images`: Image files used by the GUI.
  - `QueryProxy2`: Code for the QueryProxy.
    - `Common`: Code common to both the Leader and Node layers.
    - `Debug`: Debugging related code.
    - `Leader`: Code pertaining to the Leader layer of the QueryProxy.
    - `Mote`: Code for interacting with the Berkeley Motes.
    - `Node`: Code pertaining to the Node layer of the QueryProxy.
    - `QueryProcessor`: Query processing code used by the QueryProxy and the FrontEnd.
    - `Request`: Classes representing messages and XML parsing code.
    - `Sensors`: Code for getting sensor readings from sensors.
  - `scripts`: scripts for running simulations
  - `simTracker`: Java code for simulating tracked objects in the sensor network

5.2 QueryProxy Code Summary

- The *QueryProxy* code consists of several main parts:
- The files under "`Leader`" pertain to the leader node layer.
- The files under the "`Node`" directory pertain to the node level layer.

- The files under "`Request`" pertain to the XML messages that are sent between queryproxy node and leader layers and the frontend. Some of these classes are also used by the FrontEnd code.
- The files under "`Sensors`" are for sampling the sensors on the Sensoria nodes.
- The files under "`Debug`" define debugging macros such as: ASSERT(x), DEBUG(x). These macros are also used by the Frontend code.
- The files under "`QueryProcessor`" pertain to the execution of queries. It does things like starting and stopping queries, duplicate elimination, and tuple aggregation and processing. This code runs in the node layer, the leader layer, and the frontend.
- The code under "`Common`" includes utility functions, directed diffusion networking layers, and some common data structures. This code also runs in the FrontEnd.

5.3 QueryProxy Code Execution

5.3.1 QueryProxy2: The Main Thread

Files: `QueryProxy2.{cpp,h}`

`QueryProxy2`, the executable for the QueryProxy program, takes four arguments - a nodeID, and x, y, z coordinates. When the QueryProxy starts up, it creates a thread for the leader layer and a thread for the node layer. It creates a data structure to hold the data passed in at startup, and then it waits for the two threads to finish and the program quits.

5.3.2 NodeProcessingLayer

Files: `QueryProxy2/Node/NodeProcessingLayer.{cpp,h}`

The NodeProcessingLayer does all the work for the NodeProxy. This layer creates and starts a thread that runs a QueryPlan. The main loop of the NodeProcessingLayer waits for messages. Query related messages (starting/stopping) are forwarded to the QueryPlan. The only other messages the NodeProcessingLayer needs to handle are NodeID messages received from other cluster members during cluster creation, which are internally (not over directed diffusion) sent to the LeaderProcessingLayer.

5.3.3 QueryPlan

Files: `QueryProxy2/Node/QueryPlan.{cpp,h}`

A QueryPlan is a class that is responsible for executing queries regularly, doing some duplicate elimination, and sending out the results to the leader of the cluster. It maintains and runs all of the currently active queries that run on the node.

The main thread of this class waits on a queue listening for commands issued by its creator (either a front end, node, or leader) and also timer messages to wake itself up to process queries periodically.

Processing a new query or deleting a query involves adding or deleting from the ActiveQuery list. The regular activation of queries is done by the method '`void ProcessQuery()`', which follows this simple outline of steps:

1. Go through the ActiveQuery table and see which queries need to be run.
2. Run each of those.
3. Schedule a timer so that it can wake up to execute the next query that is due.
4. Sleep until the next query must be run.

The QueryPlan uses a DeviceManager created in NodeProcessingLayer, to get the actual tuples for the sensors involved in the queries it needs to process. The DeviceManager interacts with the code in QueryProxy2/Sensors.

The actual queries are implemented by the ActiveQuery data structure described in the next section.

5.3.4 ActiveQuery

Files: `QueryProxy2/QueryProcessor/ActiveQuery.{cpp,h}`

Internal Representation of Queries - ActiveQuery

The code in ActiveQuery handles the processing of queries. Each ActiveQuery object represents all queries that are equivalent (only difference will be the ID assigned to each query). Thus, each ActiveQuery object maintains a list of instances of that specific query.

When a query is initially posed, the ActiveQuery first analyzes the query and, if running on the FrontEnd or Leader, creates a possibly different query to ask the next lower level of nodes (Leaders and Cluster Nodes, respectively). Then the newly created query is sent to the proper destinations.

When new tuples arrive, they are delivered to the ActiveQuery object via a function call. The tuples are held in a buffer that keeps only the most recent tuple received from each cluster node.

When the query is to be processed (by the QueryPlan), ActiveQuery first retrieves all the tuples it has received from the buffer. Then, depending on the query, aggregates may be computed and the result is sent to the next higher level.

5.3.5 LeaderProcessingLayer

Files: `QueryProxy2/Leader/LeaderProcessingLayer.{cpp,h}`, `QueryProxy2/Leader/LeaderProcessingLayerHandlers.cpp`, `QueryProxy2/Leader/LeaderProcessingLayerMessages.cpp`

The implementation of functions defined in the `.h` file is split between these three `.cpp` files.

The LeaderProcessingLayer is the main class in the Leader layer. The leader layer handles clustering, aggregating tuples from the nodes in its cluster, and sending results and receiving commands from the frontend. It may also need to modify some of the queries received from the frontend before forwarding it to the nodes in its cluster.

The LeaderProcessingLayer creates a QueryPlan to handle the execution of queries. For all query related messages (new query, stop query, new tuple), the message is forwarded to the QueryPlan. The LeaderProcessingLayer handles cluster creation itself, however.

When a cluster is to be created, a leader must be elected. To do so, the LeaderProcessingLayer sends its NodeID to all other nodes in the new cluster (via the Leader->Node channel). The NodeIDs are received by the NodeProcessingLayer and are internally forwarded to the LeaderProcessingLayer. Then, after a period of time, the election is ended and the node with the greatest NodeID is declared the leader.

The LeaderProcessingLayer also handles testing operations such as retrieving node information and starting/resetting nodes.

5.4 FrontEnd Code Summary

The code for the Frontend is mainly in one directory: `FrontEnd2`

- `main.h` - contains all the main include files
- `DDFrontEnd.{cpp,h}` - contains some FrontEnd specific networking code using directed diffusion routing protocol.
- `Database.{cpp,h}` - handles writing tuples received by the FrontEnd from the queryproxies into a remote MySQL database.
- `FrontEnd.cpp` - contains the main function and starts the main thread
- `ConnectionHandler.{cpp,h}` - processes both directed diffusion connections with the cluster leaders and a TCP/IP connection with the GUI.
- `ActiveConnection.{cpp,h}`

The FrontEnd also uses the common, debug, util, QueryPlan, ActiveQuery, and Request (the XML code) files that are in the QueryProxy2 tree.

5.5 FrontEnd Code Execution

FrontEnd binds a defined socket and listens on this socket. When it receives a connection, it spawns a thread to handle this connection. That connection is processed by `ConnectionHandler::Process()`. For each connection another thread is created to dispatch the request.

Query messages – Query related operations are forwarded to a QueryPlan, as in the leader and node layers.

Tuple messages – Tuples coming from the Cluster Leaders are delivered to the QueryPlan, which is responsible for sending them back to the client.

5.6 GUI Code Summary

The code for the GUI is in one directory: `GUI`

- `ConnectionInfo.java`, `Connection.java` – handles the connection between the GUI and the FrontEnd/gateway node.
- `DatabaseConnection.java` – handles connections to an external database (such as MySQL or Postgres).
- `Debug.java` – contains the code for printing out Debug messages
- `FrontEndGUI.java` – this is the main java class with most of the component layout and interface drawing
- `Global.java` – global parameters and variables used by the other classes
- `ImageFilter.java`, `ImagePreview.java` – classes to aid in loading an image from a file into the GUI
- `JIMTable.java` – Java IMproved Table for displaying tuple results
- `Query.java` – encapsulates a query to the sensor network
- `ResultSetUtilities.java` – contains helper functions for processing results from an external database
- `SelectionArea.java` – contains all of the code drawing the map, e.g. the nodes, the cluster, the detections, the tracks, etc.
- `SQLQuery.java` – encapsulates a query to an external database
- `XML.java` – contains all of the XML parsing code

5.7 GUI Code Execution

When the user clicks “Connect”, the GUI will form a TCP/IP connection to the FrontEnd node. The connection is verified to be valid and then the user is allowed to start querying.

The first query must be a status query to retrieve the position and status of each node. This is necessary since the nodes must be drawn on the map to allow the user to selected areas on the map to query further.

Once the status query is finished, all other types of queries are available to be run. The user can select pre-defined queries to run or can enter ad-hoc queries for the sensor network. The user can also define a box area to run the query or select a list of nodes that should respond.

XML query messages are sent from the GUI to the Frontend node. Tuple responses are displayed in a table for perusal by the user as soon as they arrive. Certain types of queries may also display information on the map, e.g. detections and tracks.

An interface for interacting with an external database is also provided. It is possible to output the results of queries to a MySQL or Postgres database for archival and the GUI can query those results.

5.8 SimTracker Code Summary

The code for the SimTracker is in one directory: `SimTracker`. Code is not shared with the main querying GUI, since the functionality is very different.

- `ConnectionInfo.java`, `Connection.java` – handles the connection between the SimTracker and the nodes in the network.
- `Debug.java` – contains the code for printing out Debug messages
- `DetectionForwarder.java` – handles sending the simulated detections to the correct node.
- `Global.java` – global parameters and variables used by the other classes
- `ImageFilter.java`, `ImagePreview.java` – classes to aid in loading an image from a file into the SimTracker
- `Path.java` – encapsulates the data for a path that a simulated target travels through the sensor network.
- `SelectionArea.java` – contains all of the code drawing the map, e.g. the nodes, the tracks, the targets, etc.
- `SimTracker.java` – this is the main java class with most of the component layout and interface drawing
- `Target.java` – encapsulates the simulated target data
- `XML.java` – contains all of the XML parsing code

5.9 SimTracker Code Execution

The SimulatedTracker allows users to draw simulated targets in the sensor network for testing. It does this by forming a direct connection to each node in the network. The user can then draw a path on the map and choose a target to follow that path. When the target is within a certain distance of a node, the SimTracker sends a simulated detection directly to the sensor node (see later section on GUI/SimTracker walkthroughs).

The idea is that another user will be querying the network for detections with the main GUI. When the simulated detections arrive at the intended node, the detection is processed and handled in the exact same manner as real detections. This enables users to test the sensor network for problems when running detection or tracking queries.

6 Parsing XML Messages and Requests

6.1 Reading XML Messages with the Request class

The XML messages used by the QueryProxy are read using the `Request` class, located in `QueryProxy2/Request` directory. To parse the XML message, the static function `Request::fromString` is called with the XML message, the type of node that sent the message, and the NodeID of the node which sent the message. This function will return a pointer to a `Request` object if the XML parsing is successful, or `NULL` otherwise.

The `Request::GetType()` function determines what type of message was received and returns one of the TYPE_* constants defined in `Request.h`, e.g. `CREATE_CLUSTER`, `QUERY`, `TUPLE`, `QUERY_DELETE`, etc. Most of the message types have some data associated with them, which can be obtained via the `Request::GetData()` function. This function returns a pointer to a `RequestData` object. The data for each type of message is stored in a subclass of `RequestData`, e.g. the `CreateClusterData`, `QueryData`, `NodeIDData`, and `Tuple` classes.

6.2 Extracting Sensor Data from a Tuple

A `Tuple` consists of a linked list of `SensorFields`. Each `SensorField` in a `Tuple` must have a unique sensor and field. The value of a specific `sensor.field` is obtained by calling `Tuple.GetData()` with the sensor and field names and an optional aggregate.

For example, for the query 'SELECT MAX(light.value) FROM light' one would call 'tuple->GetData("light", "value", "max")' to obtain the value of MAX(light.value).

To get a list of SensorFields in a Tuple, the linked list of SensorFields must be traversed. To do so, GetFirst() is called to get the first SensorField. Each SensorField is then examined with following functions:

- GetSensor() – returns the sensor name
- GetField() – returns the field name
- GetAggregate() – returns the aggregate function (if any)
- GetData() – returns a pointer to the data stored in the DataType class

GetNext() obtains the next SensorField in the list. Note that the SensorFields in a Tuple may or may not be in any particular order. The actual data is stored in the DataType object associated with each sensor and there are three member functions named GetInt(), GetDouble(), and GetString() that return the sensor value depending on the type of the data.

For more details, the SensorField class is located in the SensorField.{cpp,h} files and the DataType classes are located in DataType.{cpp,h}.

6.3 Creating New XML Messages

To create a new XML message, one must write a subclass of RequestData to hold an internal representation of that data, and add functions to read and write the XML message. A short outline of this procedure follows.

1. First, add a new TYPE_ constant to the Request.h file.
2. Next, modify Request::ReadXMLType in RequestXMLRead.cpp to recognize the new XML message. This function, when supplied with the new XML message, should return the new TYPE_ constant.
3. Request::ReadXMLMessage in RequestXMLRead.cpp must be modified to have a new case statement for the new TYPE_ constant. It should call a function which will handle the parsing of the new XML message. This function should return the new subclass of RequestData that represents the message, or NULL if the message could not be parsed.
4. Finally, add a new case statement to Request::WriteXML in RequestXMLWrite.cpp for the new TYPE_ constant. It should call a function which handles the writing of the XML message. It is assumed in this function that the RequestData subclass is valid and can be properly written as XML.

7 GUI Walkthrough

7.1 Config Panel

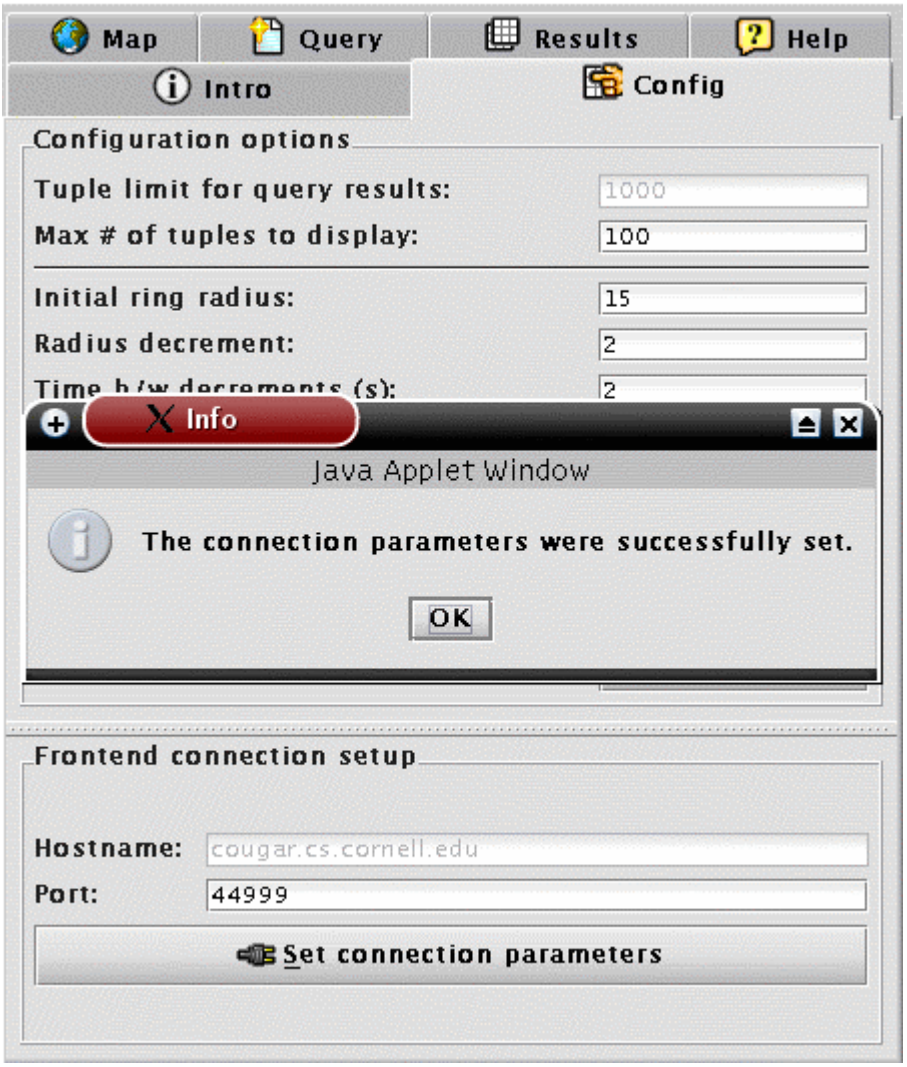

Figure 6: Config Panel Setup

The Config panel is used for changing certain query and display options as well as setting up the connection to the gateway/frontend node.

Connection parameters are set when the GUI is started, but you can recheck them:

- Click on the "Set connection parameters" button to test the network connection to the sensor network.
- You should get a message verifying that the connection to the sensor network is working.

7.2 Map Panel

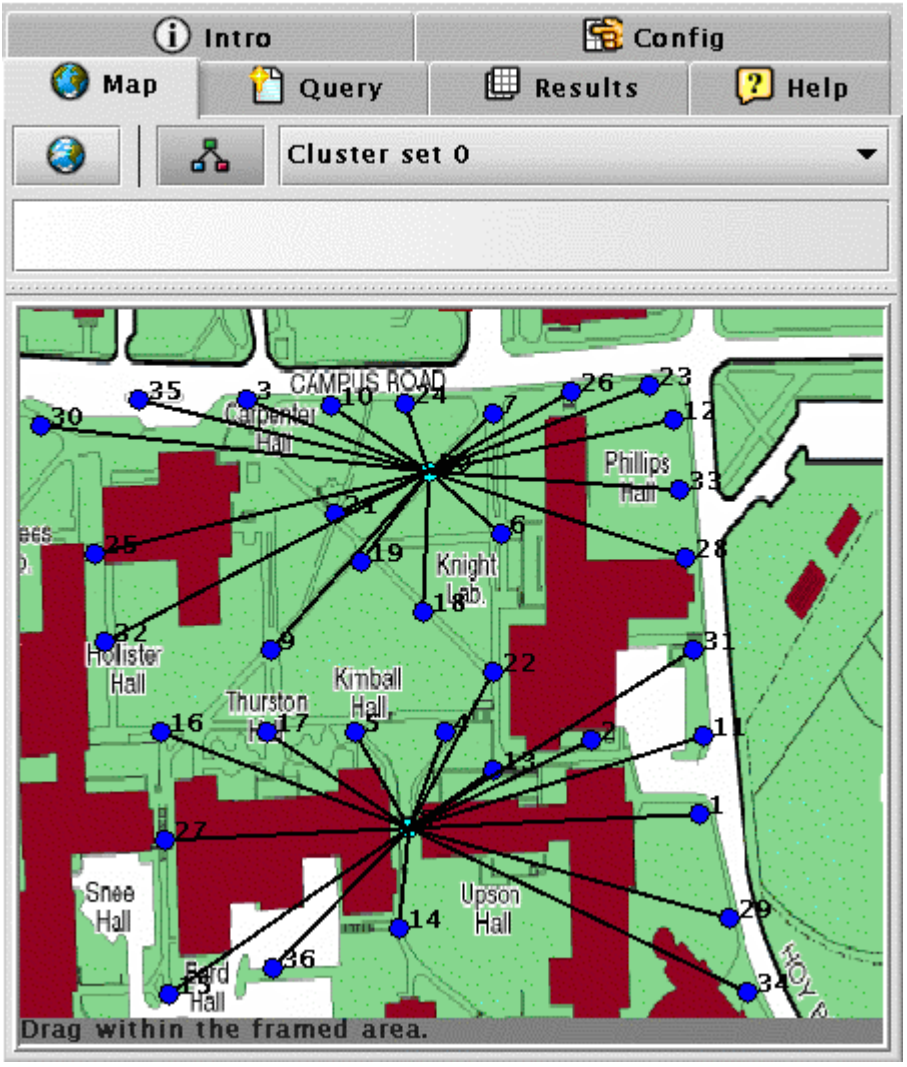

Figure 7: Map Panel – Show Clusters

From left to right, the top buttons are:

- Update map with current node info
- Show/Hide cluster info on the map
- Select cluster set for display
- Select query for display

Start using the GUI:

1. Click on the "Update Map" button (upper left corner) to retrieve node positions and statuses.
2. Click on "Show Clusters" button to toggle the display to show cluster information.

3. If no cluster information appears, the sensor network was not initialized properly and a system administrator must re-initialize it.
4. Cluster Set 0 is the default cluster set and must appear for any queries to run.

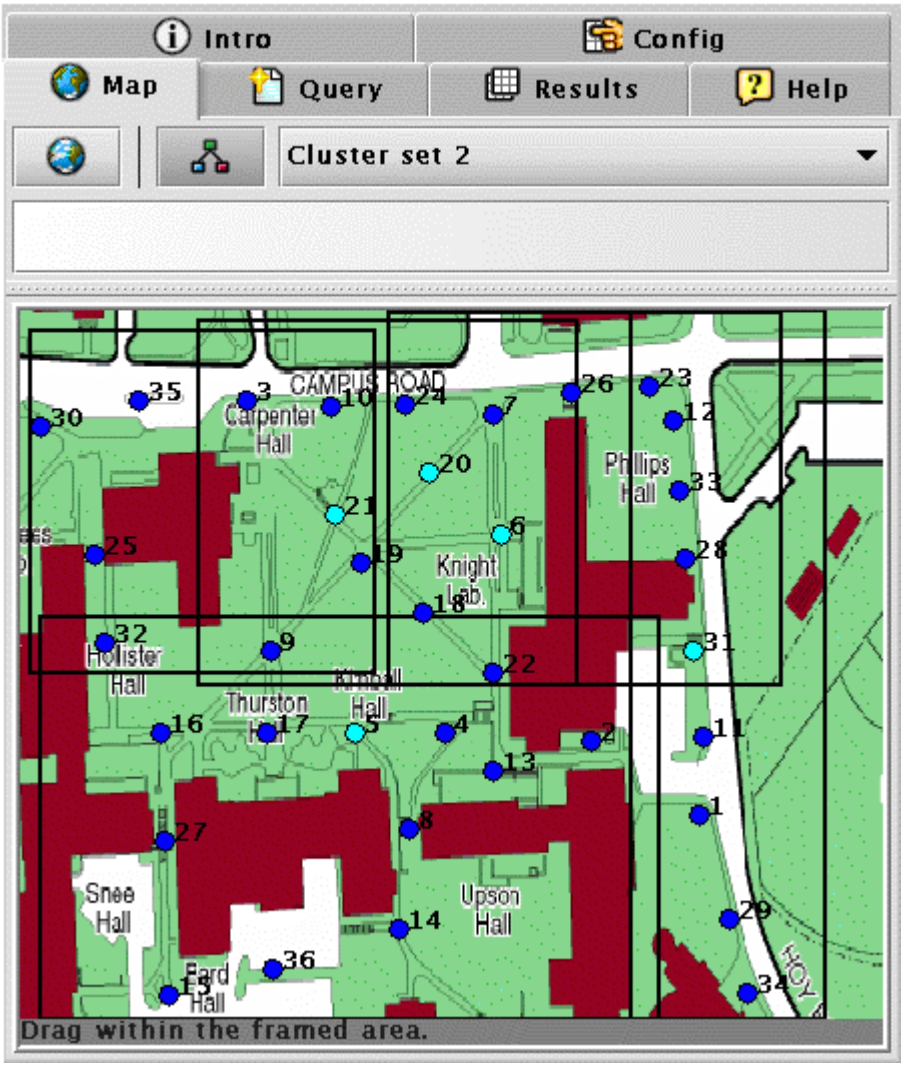

Figure 8: Map Panel – Show Overlapping Clusters

5. Select a different cluster set to display.
6. In this example, Cluster Set 2 has several overlapping clusters.

7.3 Query Panel

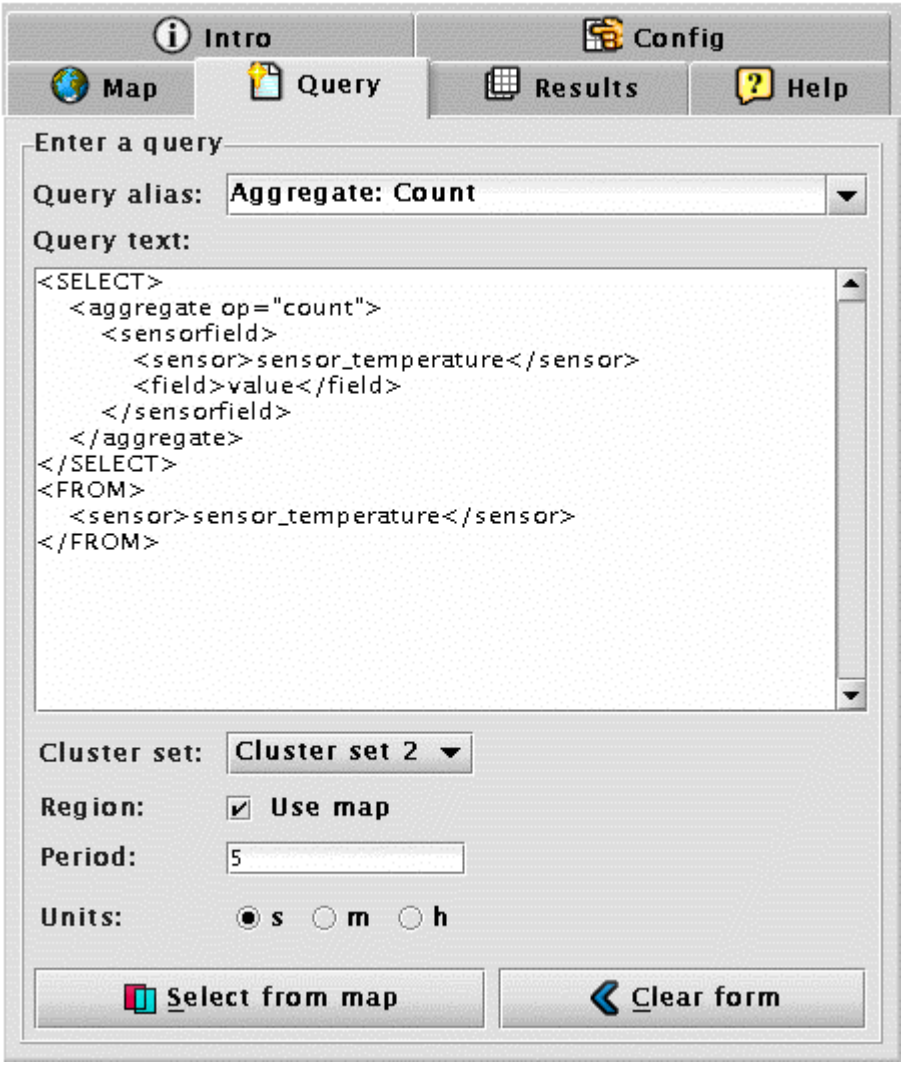

Figure 9: Query Panel

Running an aggregate query:

1. Select a query from the drop down box.
2. This example is a simple aggregate over the temperature sensor. Since each node has one (simulated) temperature sensor, the query should return the number of active nodes.
3. Fill in a period value. 5 seconds is used for this example.
4. Click on "Use map" for selecting a query region on the map. If the map is not used, all nodes are selected by default.
5. Click on "Select from map" to switch to the map panel.

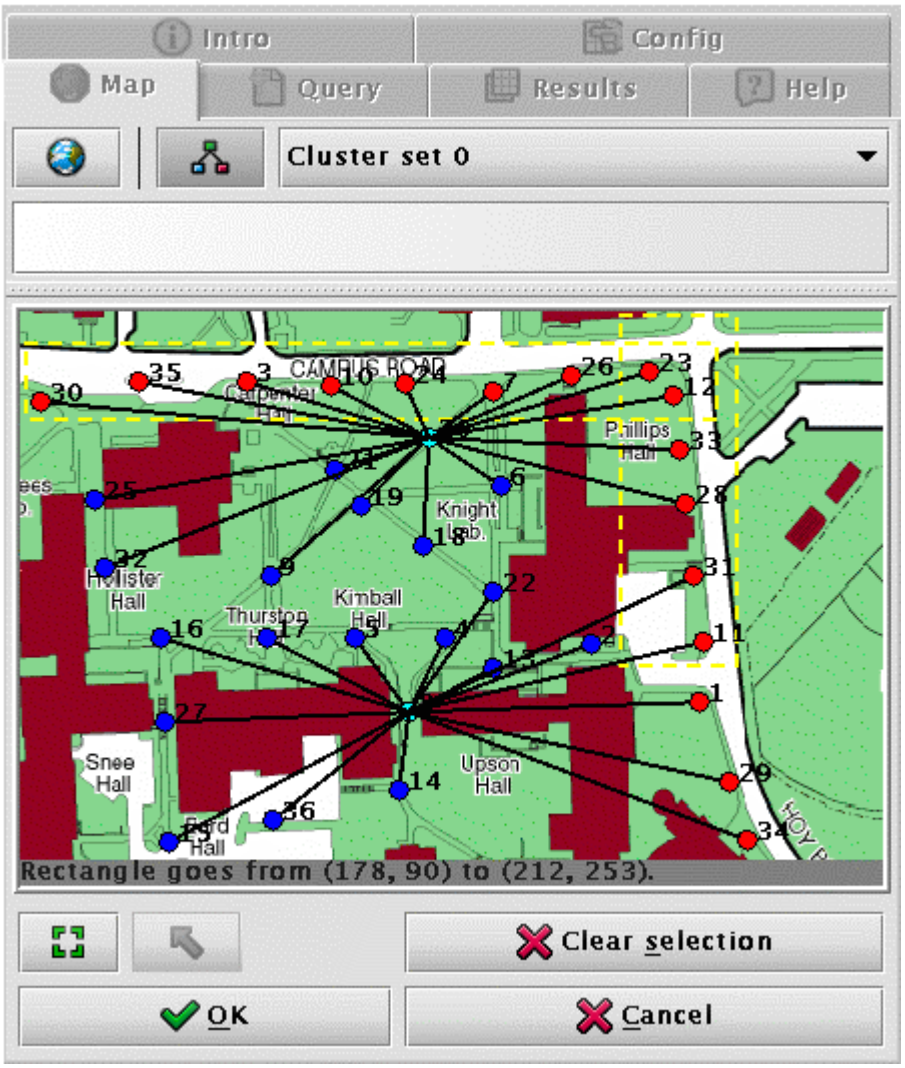

Figure 10: Query Panel – Select Area

6. Nodes are selected with box regions or by individual selection.
7. Click on the Box button on the bottom left.
8. In this example, two box region are drawn on the map to correspond to outer perimeter of nodes
9. Click on the Arrow button on the bottom left.
10. Select the remaining perimeter nodes by clicking on them individually.
11. All selected nodes will change color to red.
12. A different cluster set can be chosen for the query at the top of the panel.
13. Click on "OK" to start the query.

7.4 Results Panel

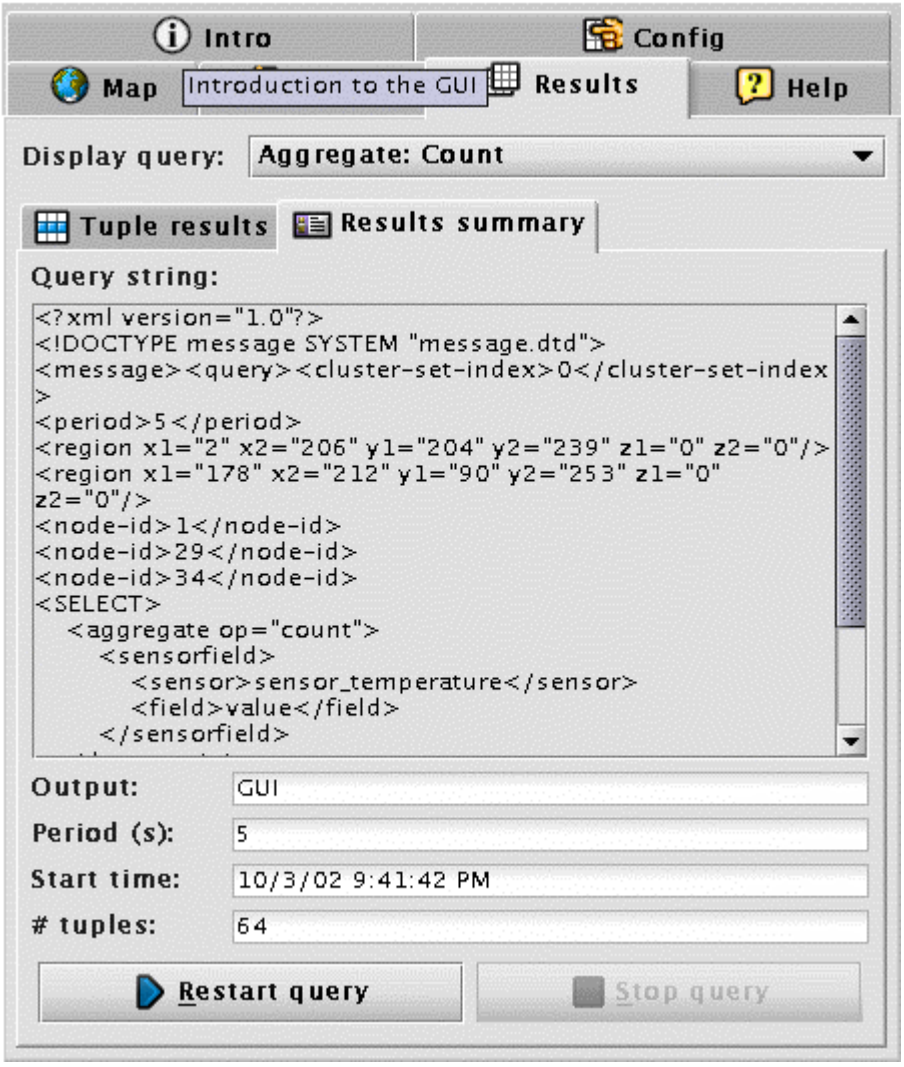

Figure 11: Results Panel

- Individual tuples from the query will appear in the Results panel.
- Periodic queries will usually take twice the period time to return the first tuples.
- Different query results can be selected from the drop down box for display.

- A summary of the query and its results is also available.
- On the Summary Results panel you can also stop and restart different queries by using the buttons on the bottom.
- The full XML message that is sent to the gateway node is displayed here as well.
- Notice the list of regions and nodeids associated with the query.

7.5 Detection Queries

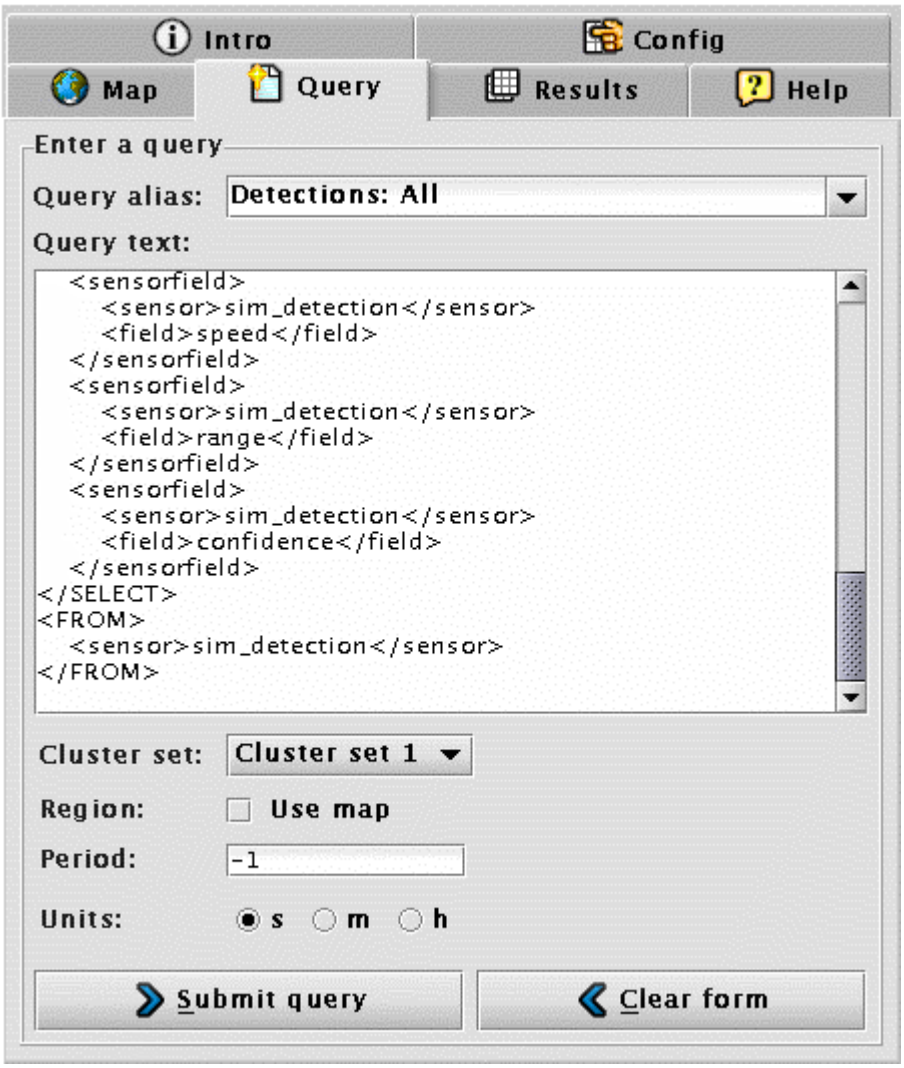

Figure 12: Detection Queries

1. Detection queries are treated differently than other ad-hoc queries.
2. Select "Detections: All" from the drop down box.
3. The period for detection queries is set to -1 to indicate that tuples are returned only when a valid detection has occurred
4. For this query, we are not using the map, so we must select a cluster set in this panel. This example uses Cluster set 1.
5. Click "Submit query" to start the query.
6. Compare the XML message for the detection query with the previous query.

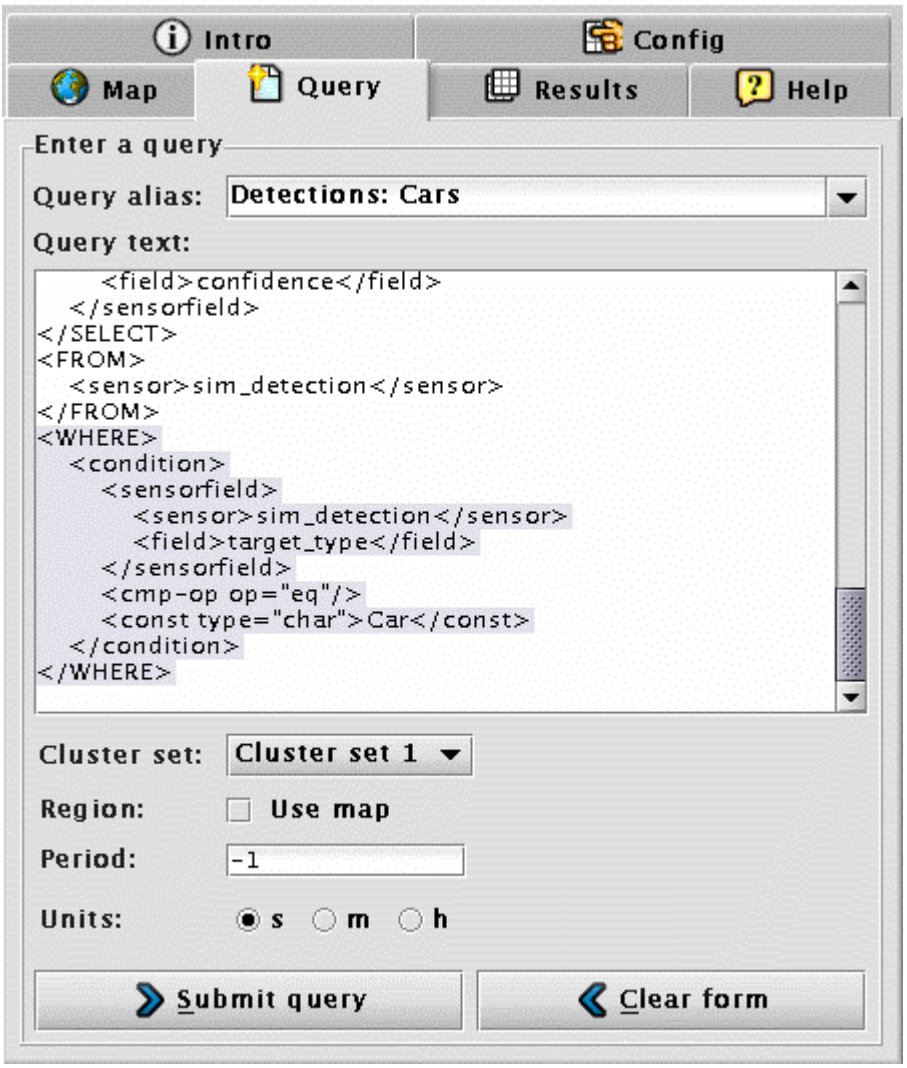

Figure 13: Detection Queries – Filter by Type

7. Select "Detections: Car" from the drop down box.
8. Note that the XML for the query is the same as for "Detections: All" except for an extra "WHERE" clause.
9. The different possible target types are:
   - Unknown
   - Pedestrian
   - Bike
   - Car
   - Tank
   - Helicopter
   - Airplane
10. By changing the "WHERE" clause, you can form a query to select by any type of target.

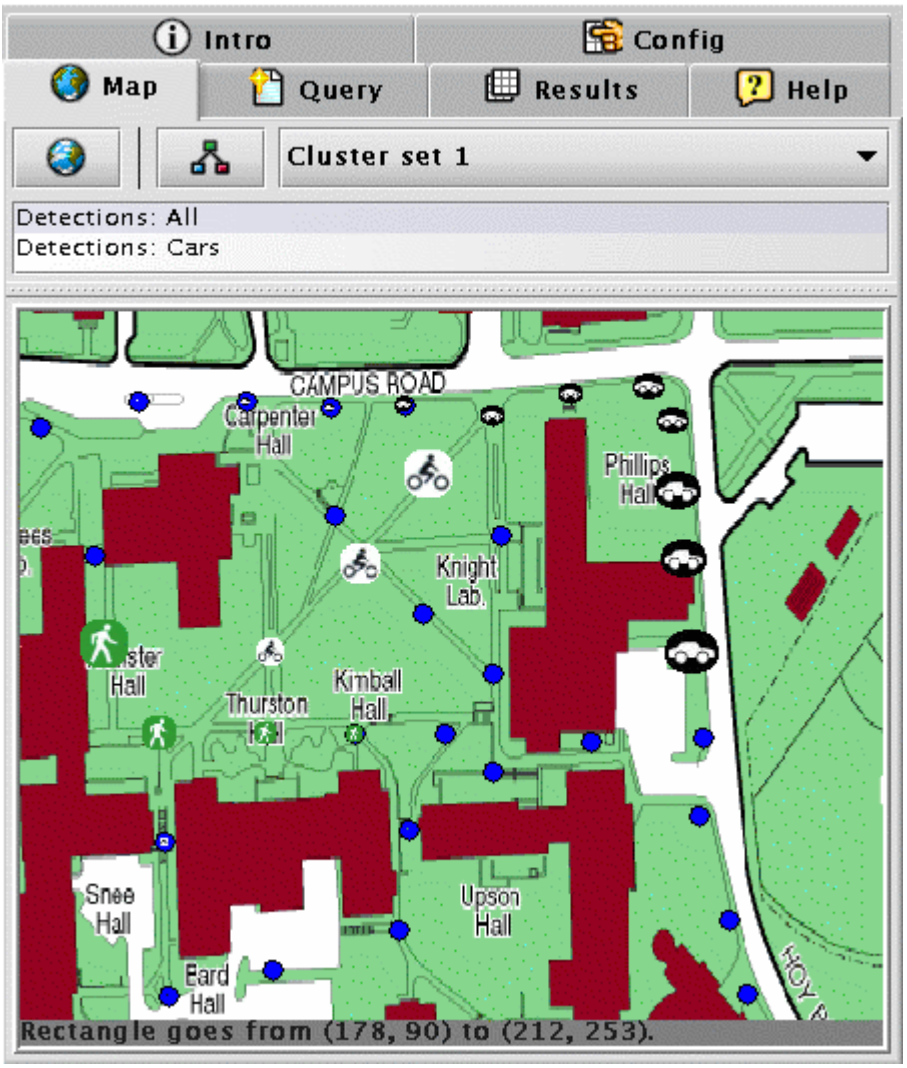

Figure 14: Detection Queries – Show Results

11. On the Map panel, the detection queries will appear in the top selection box.
12. Use the SimTracker to generate simulated detections for your query.
13. After the simulated targets are started, detection results will begin to appear.
14. Different target types display with a different icon and fade over time.
15. You can multi-select which queries to display detections for.
16. This example shows all of the detections.

7.6 Tracking Queries

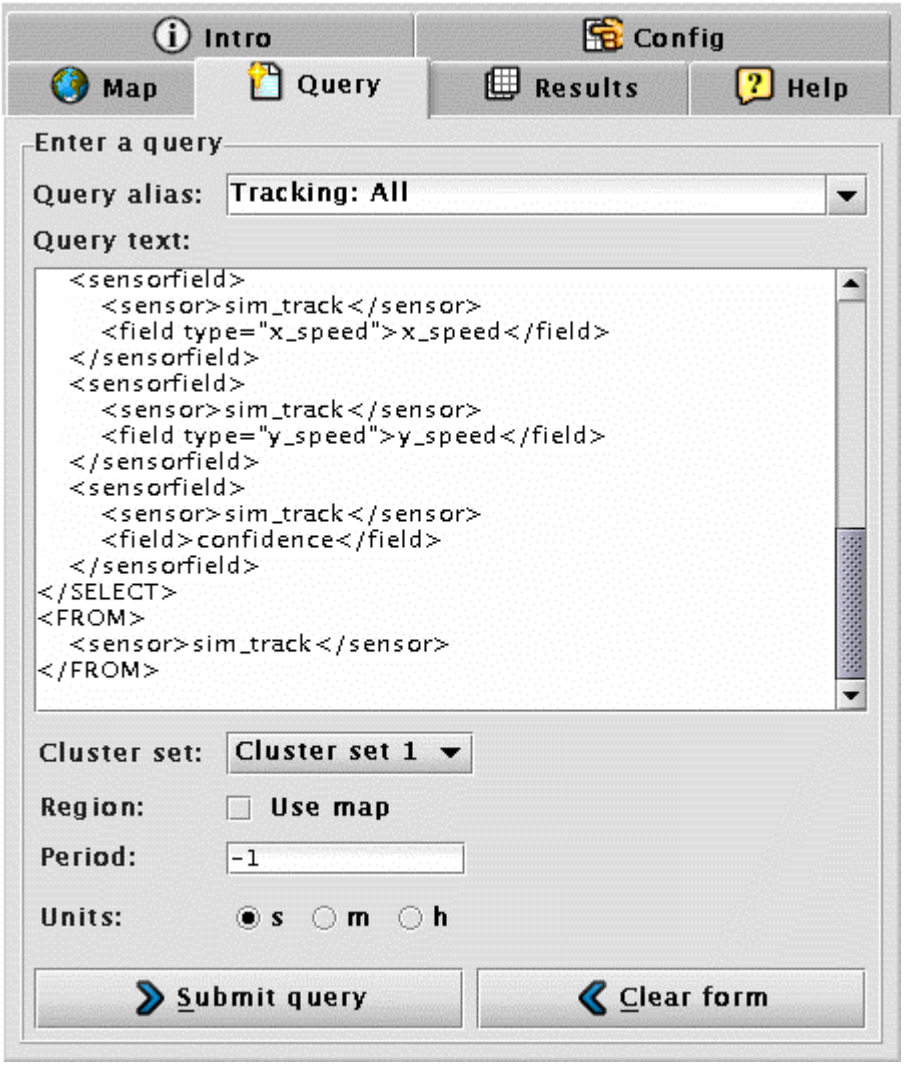

Figure 15: Tracking Queries

1. Tracking queries run much like detection queries.
2. The period should also be set to -1.
3. In this example, we are running matching tracking queries for the detection queries, but tracking queries are completely independent of detection queries.
4. Start a query for all tracks and one for only car tracks.

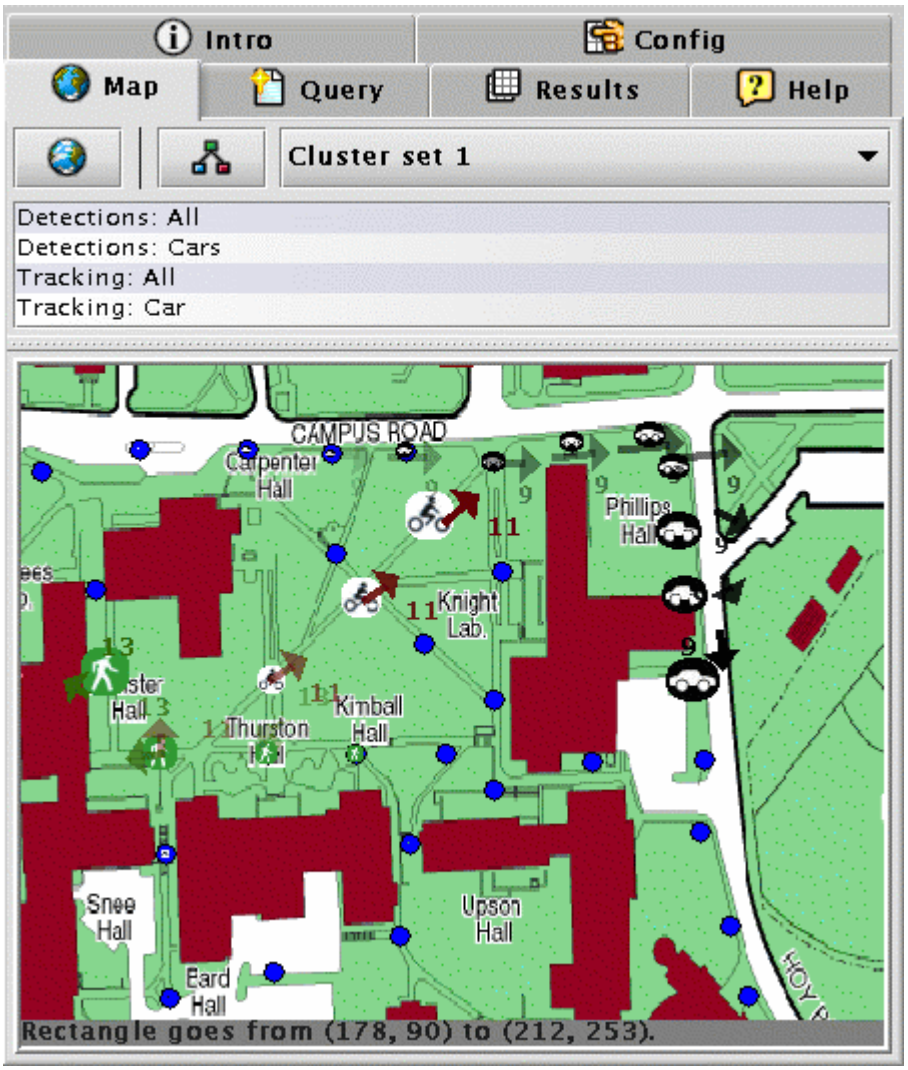

Figure 16: Tracking Queries – Show Results

5. Run the SimTracker simulation again.
6. Track results are displayed as colored arrows with the associated trackid.
7. The arrows fade out over time.
8. Multiselect to display all tracks and all detections.
9. With this display, you can see the actual detections along with tracks.

8 SimTracker Walkthrough

8.1 Map Panel

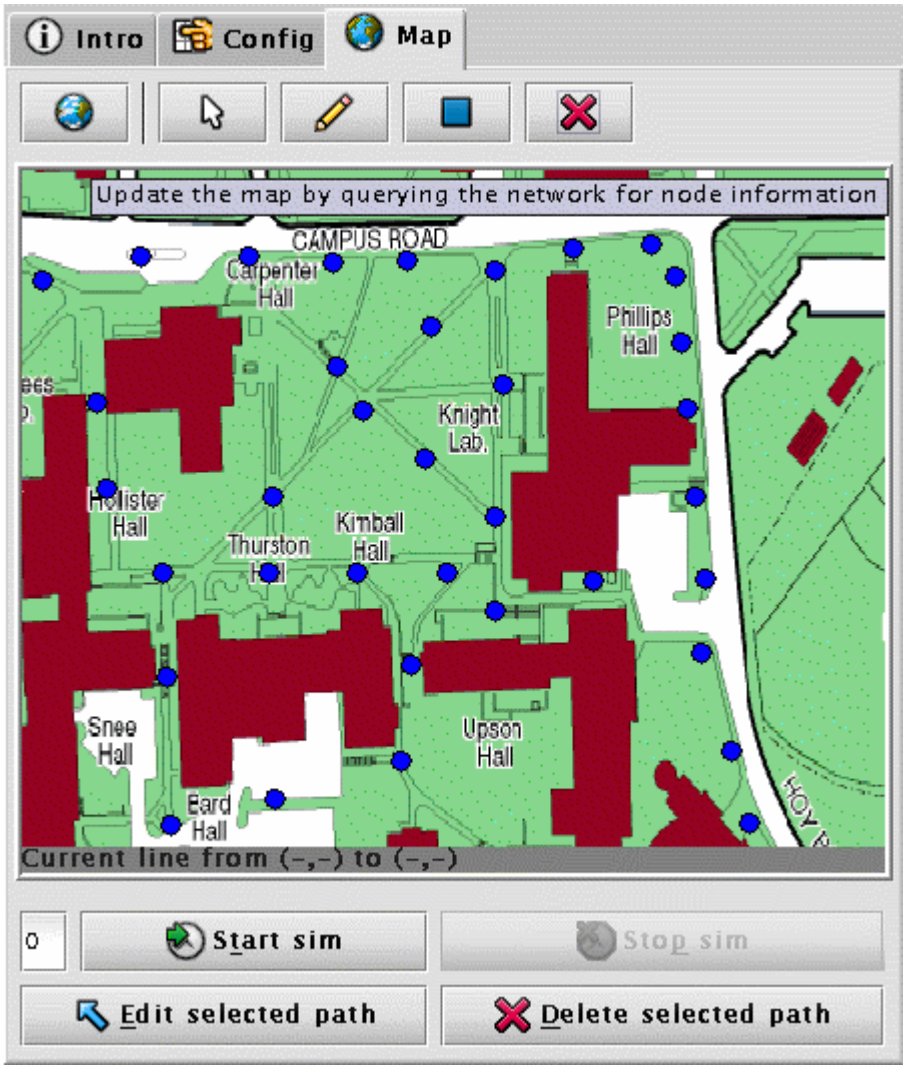

Figure 17: SimTracker Map Panel

From left to right, the top buttons are:

- Update map with current node info
- Select paths by clicking
- Draw a new path on the map
- Save current path being drawn
- Clear current path being drawn

8.2 Drawing Simulated Paths

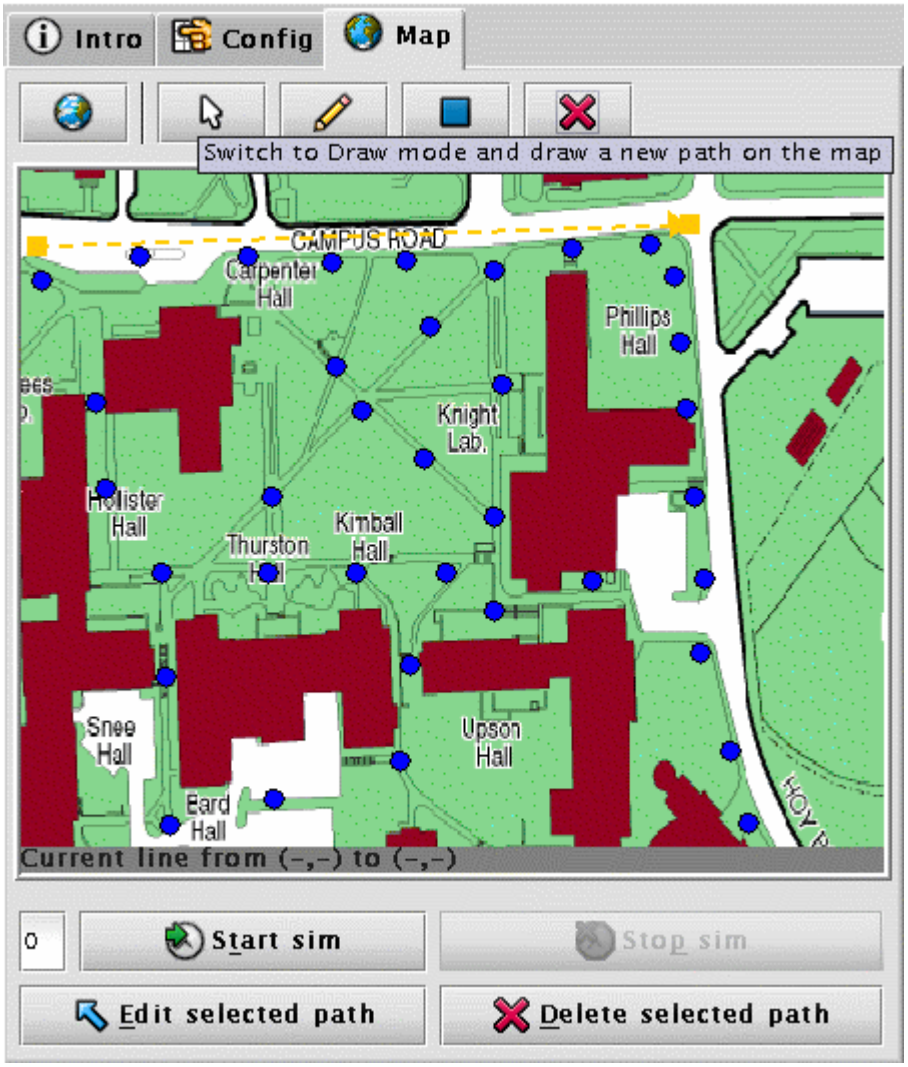

Figure 18: Drawing Paths

1. Click on the "Add new path" (pencil) icon.
2. Click on the map to mark the start of the path.
3. Click at the end of the line segment that the target should traverse.
4. New paths being drawn will appear as dashed.
5. In this example, I clicked on the upper left of the map and then followed the road to the intersection at the upper right.

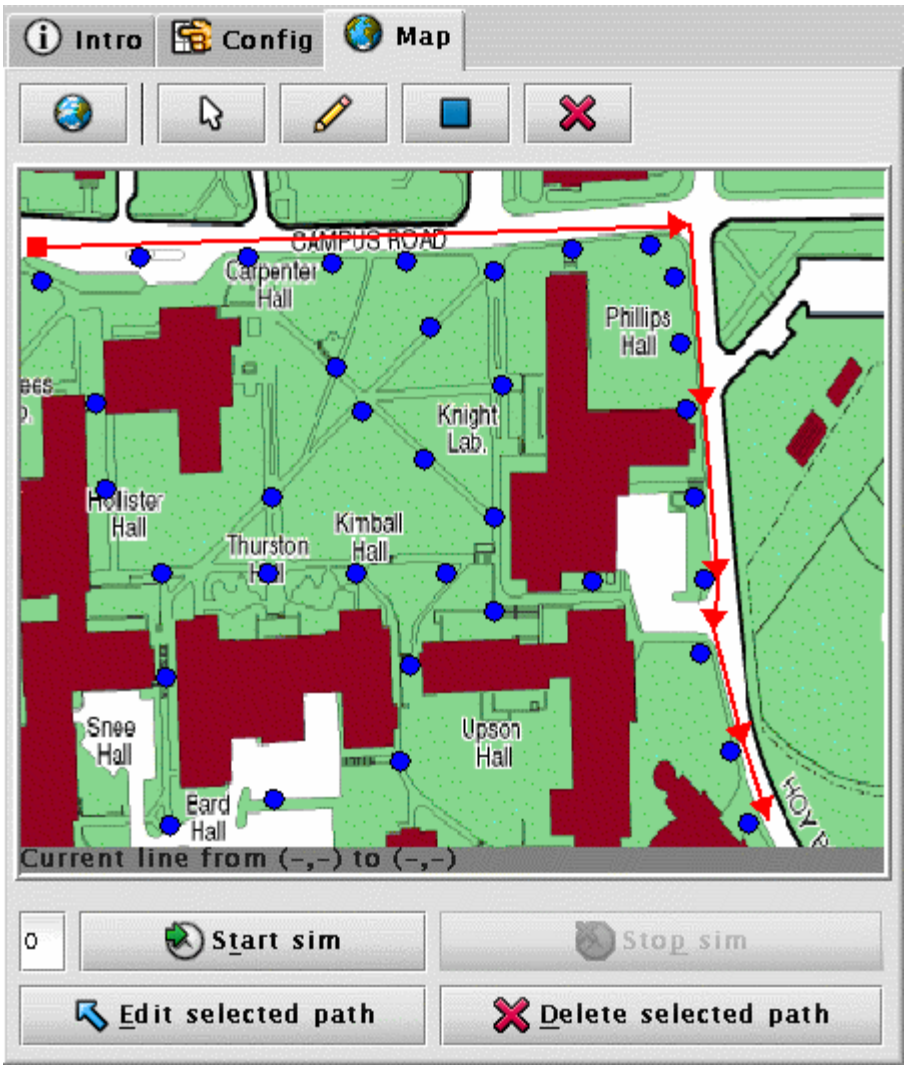

Figure 19: Saving Paths

6. Add more line segments to make a right turn at the intersection and follow the road to the edge of the map.
7. Click on the "Save Path" button (the blue box).
8. The path lines will become solid, indicating that they are saved.
9. Newly saved paths are marked as selected, so the path appears red.

8.3 Adding Simulated Targets

Figure 20: Adding Targets

1. Click on "Edit selected path" on the bottom button panel. The last path drawn should already be selected in red.
2. The target list for the path will appear. Since this is a new path, the list will be empty.
3. Click on "Add target".
4. The "Add target" dialog will appear. Select "Car" from the drop down box.
5. The target parameters can be changed from the defaults. Mouseovers on the different parameters will display more information.

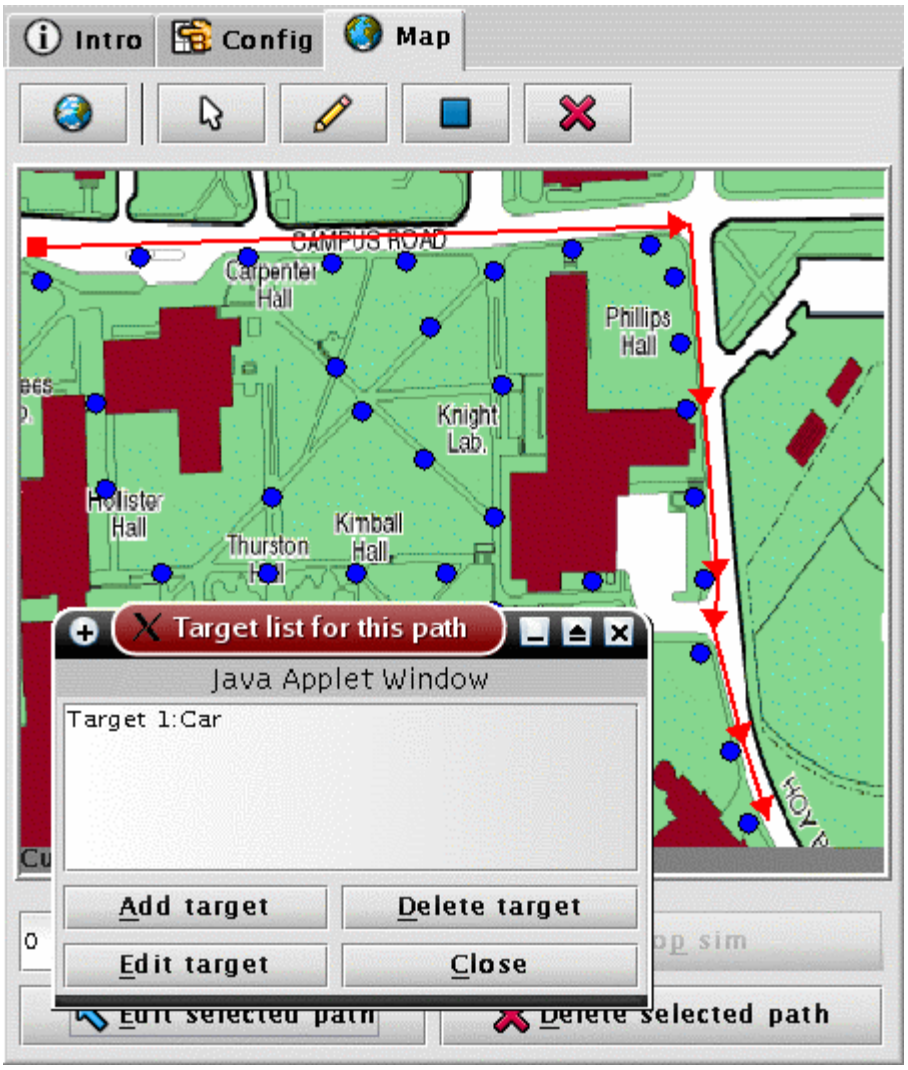

Figure 21: Editing Targets

6. After you click "OK", the car will appear in the target list for this path.
7. You can add more targets, edit old targets, or delete targets from the same window.
8. Once you are done adding targets, click "Close" to return the main Map window.

8.4 Running A Simulation

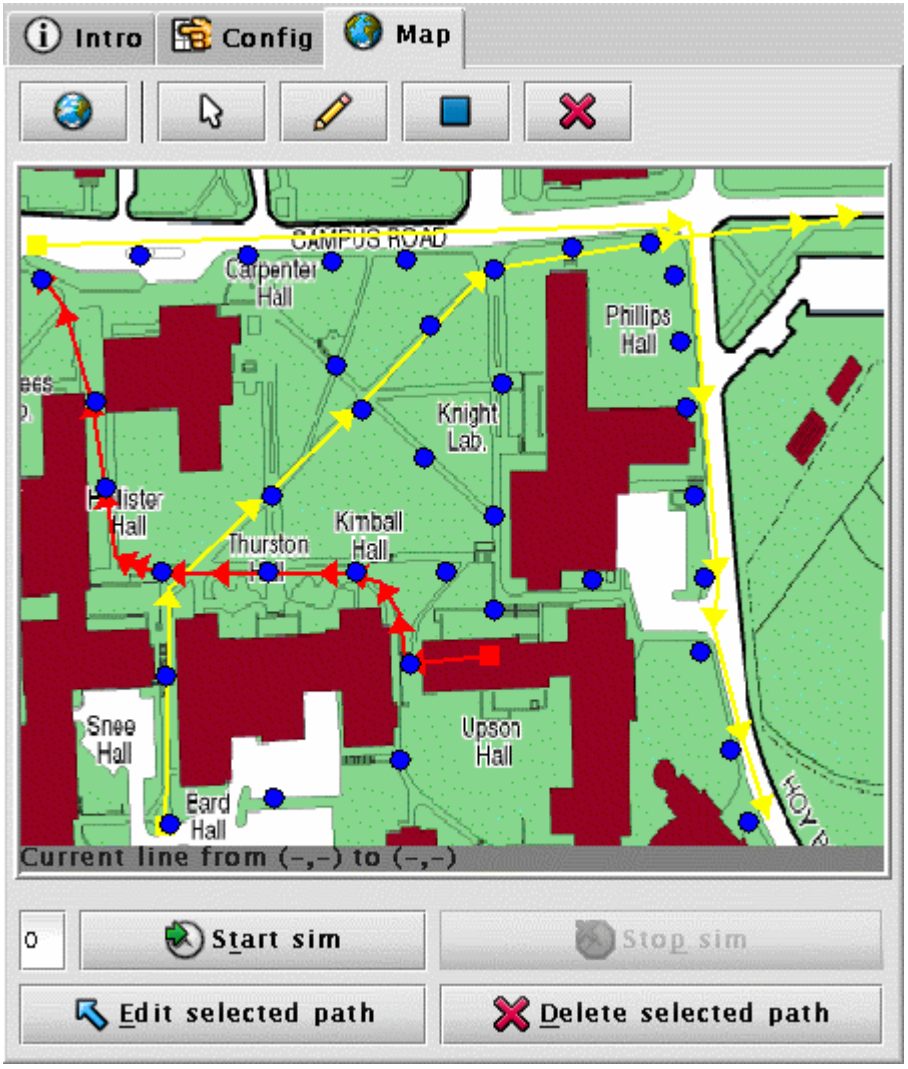

Figure 22: Simulated Paths

In this example, the targets were created as:

1. A car driving along the top road and then turning right.
2. A bike coming from the bottom left and moving diagonally through the map.
3. A pedestrian starting in the Upson Hall building and then walking towards the upper left.

The pedestrian path is highlighted in red since it's selected. The other paths are in nonselected yellow.

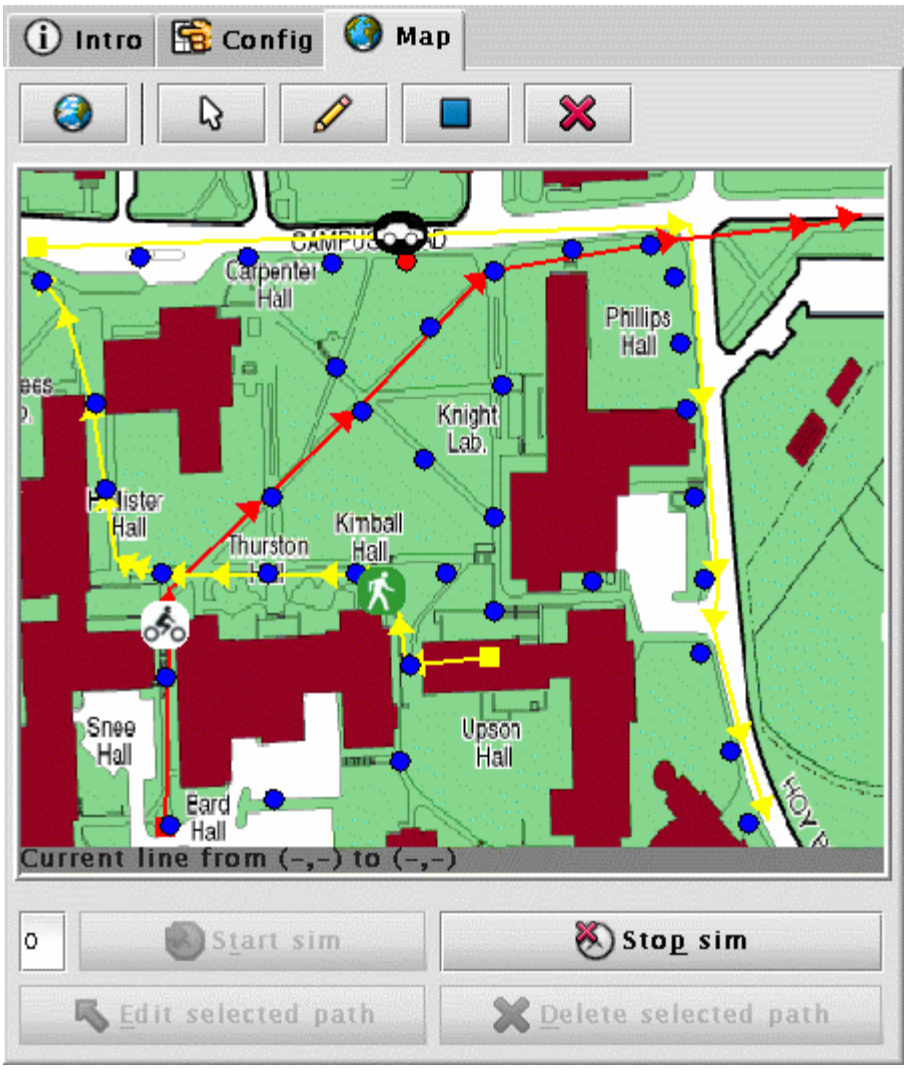

Figure 23: Running a Simulation

1. Now we're ready to start the simulation. The box next to the "Start sim" button sets the number of times to repeat the simulation. Zero means the simulation only runs once.
2. Click on "Start sim" to begin the simulation.
3. In this example, all three targets start at the same time and move along their respective paths.
4. When a target is close enough to a node to set off a detection, the node will turn red.
5. The simulation ends after all of the targets have reached their destinations.
6. If the "Repeat Sim" value is more than zero, the simulation will restart and all targets will begin again at the start of their paths.

9 Significant Accomplishments Over the Lifetime of the Contract

9.1 Software

We have developed two version of the Cougar query proxy: Cougar V1.0 version for the first WINS NG architecture running Windows CE. In January 2001, a Linux-based architecture for the second generations of the WINS NG nodes was introduced, and we the developed Cougar V2.0 query proxy on top of Linux for the WINS NG architecture 2.0. Cougar V2.0 performs in-network query evaluation and distributed query processing. Cougar V2.0 is integrated with the ISI-East Routing Layer for direct deployment on the WINS 2.0 hardware, and with BAE signal processing to obtain sensor detections. We also created interfaces to several sensors for direct reading of sensor values into the Cougar System.

We supplied integration support and software to University of Maryland, Rutgers University, BAE Systems, and ISI-East for ongoing integration of their work into our system. Rutgers has integrated their prediction-based query modification into the Cougar system and they showed a demo of their integration at the PI meeting in November. We developed an interface between query processing and tracking through a SensIT working group. The working group consists of Richard Brooks (PSU), Feng Zhao (Xerox Parc) and Johannes Gehrke (Cornell).

An online demo of our software is still available at http://cougar.cs.cornell.edu.

9.2 Demonstrations

We demonstrated the Cougar System at every PI meeting, and we also demonstrated the Cougar system during the field experiment in 29 Palms in California. This first version of Cougar was demonstrated to the DARPA Directorate in 2000, and we also demonstrated this version of Cougar at the Intel Continuum Computing Conference in March 2000.
We demonstrated the Cougar system in 29 Palms during the field experiment SITEX 2002 in November 2001.

We also demonstrated the Cougar System at the 2002 ACM Sigmod International Conference on Management of Data to an audience of over 300 attendees in June 2002.
For the final PI meeting in November 2002 in Boston, we also integrated with BAE Systems and ISI East for a subgroup-demo, and with University of Maryland for a complete demo. In addition, the Cougar platform was used by several groups as a scalable middleware infrastructure for distributed query processing, and thus it built the foundation for the work of several other groups in the project. We participated in several live test-bed demos at the PI meeting in Boston in November 2002, including an

integration with ISI-East (new GUI), ISI-West (diffusion routing), BAE (signal processing). Besides diffusion routing, our software is the most widely used component within the SensIT program.

We demonstrated the complete Cougar system during an evening demo session at the PI meeting in Boston in November 2002. This included superset of the functionality that we promised to deliver in the grant, including several integrations with other groups in the SensIT program. In addition, we also showed Mini-Cougar, a version of Cougar on the Berkeley motes.

9.3 Publications

- Philippe Bonnet, J. E. Gehrke, and Praveen Seshadri. Querying the Physical World. IEEE Personal Communications, Vol. 7, No. 5, October 2000, pages 10-15. Special Issue on Smart Spaces and Environments.
- Philippe Bonnet, J. E. Gehrke, and Praveen Seshadri. Towards Sensor Database Systems. In Proceedings of the Second International Conference on Mobile Data Management. Hong Kong, January 2001.
- Zhiyuan Chen, J. E. Gehrke, and Flip Korn. Query Optimization In Compressed Database Systems. In Proceedings of the 2001 ACM Sigmod International Conference on Management of Data, Santa Barbara, California, May 2001.
- J. E. Gehrke, Flip Korn, and Divesh Srivastava. On Computing Correlated Aggregates Over Continual Data Streams. In Proceedings of the 2001 ACM Sigmod International Conference on Management of Data, Santa Barbara, California, May 2001.
- Yong Yao and J. E. Gehrke. The Cougar Approach to In-Network Query Processing in Sensor Networks. Sigmod Record, Volume 31, Number 3, September 2002.
- Anton Faradjian, J. E. Gehrke, and Philippe Bonnet. GADT: A Probability Space ADT For Representing and Querying the Physical World. In Proceedings of the 18th International Conference on Data Engineering (ICDE 2002), San Jose, California, February 2002.
- Alin Dobra, Minos Garofalakis, J. E. Gehrke, and Rajeev Rastogi. Processing Complex Aggregate Queries over Data Streams. In Proceedings of the 2002 ACM Sigmod International Conference on Management of Data, Madison, Wisconsin, June 2002.
- Francis Chu, Joseph Halpern, and J. E. Gehrke. Least Expected Cost Query Optimization: What Can We Expect? In Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2002). Madison, Wisconsin, June 2002.
- Wai Fu Fung, David Sun, and J. E. Gehrke. COUGAR: The Network is the Database. In Proceedings of the 2002 ACM Sigmod International Conference on Management of Data (SIGMOD 2002), Madison, Wisconsin, June 2002. Demo description.

- Yong Yao and J. E. Gehrke. Query Processing in Sensor Networks. In Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR 2003), Asilomar, California, January 2003.
- Tobias Mayr, Philippe Bonnet, J. E. Gehrke, and Praveen Seshadri. Leveraging Non-Uniform Resources for Parallel Query Processing. To appear in Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003). Tokyo, Japan, May 2003.
- Abhinandan Das, J. E. Gehrke, and Mirek Riedewald. Approximate Join Processing Over Data Streams. To appear in Proceedings of the the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD 2003). San Diego, CA, June 2003.
- Rohit Ananthakrishna, Abhinandan Das, J. E. Gehrke, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. Efficient Approximation of Correlated Sums on Data Streams. IEEE Transactions on Knowledge and Data Engineering, Vol. 15, No. 3, May/June 2003, pages 569-572.

10 Final Status On Each Of The Tasks

Task 1: A database query processing system that accepts queries/programs specified in object-relational SQL, and executes them over a network of sensor and actuator devices.

The Cougar system for WINS NG1.0 was delivered to BBN. The second generation of the Cougar System for WINS NG2.0 has been developed and is complete. A prototype of the Cougar System for WINS NG2.0 has been successfully demonstrated in the desert at SITEX02 in Twentynine Palms, California; a compete version of the Cougar System has been shown at the PI meeting in Boston during live demos at the BBN testbed

Task 2: An implementation of a query fragmentation mechanism that optimizes these placement of fragments maximizing proximity to sensor data sources, but constrained by the resource limitations of the sensor nodes

Cougar 2.0 implements distributed aggregation operators and places them on nodes inside the network. Cougar also implements a triggering mechanism that can trigger events from inside the network.

Task 3: A resource management layer that enables the collection of dynamic statistics on resource usage for feedback into execution / optimization.

We have designed a prototype adaptive query processing layer, and an initial version is implemented in the latest Cougar Software. A limiting factor was the time commitment required to integrate with other SensIT components, as we were designated as a core components and were integrating with several other partners in the SensIT program.

11 Percent of Technical Completion

Programmed ___100__ % Actual _100___ %

12 Appendix

Attached to this final report are four selected papers that are representative of the work in our effort:

- Philippe Bonnet, J. E. Gehrke, and Praveen Seshadri. Querying the Physical World. IEEE Personal Communications, Vol. 7, No. 5, October 2000, pages 10-15. Special Issue on Smart Spaces and Environments.
- Philippe Bonnet, J. E. Gehrke, and Praveen Seshadri. Towards Sensor Database Systems. In Proceedings of the Second International Conference on Mobile Data Management. Hong Kong, January 2001.
- Anton Faradjian, J. E. Gehrke, and Philippe Bonnet. GADT: A Probability Space ADT For Representing and Querying the Physical World. In Proceedings of the 18th International Conference on Data Engineering (ICDE 2002), San Jose, California, February 2002.
- Yong Yao and J. E. Gehrke. Query Processing in Sensor Networks. In Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR 2003), Asilomar, California, January 2003.

We also attached the demo report for one of the Cougar demos that we gave:

- Wai Fu Fung, David Sun, and J. E. Gehrke. COUGAR: The Network is the Database. In Proceedings of the 2002 ACM Sigmod International Conference on Management of Data (SIGMOD 2002), Madison, Wisconsin, June 2002. Demo description.

Appendix A:

Abstract

In the next decade, millions of sensors and small-scale mobile devices will integrate processors, memory, and communication capabilities. Networks of devices will be widely deployed for monitoring applications. In these new applications, users need to query very large collections of devices in an *ad hoc* manner. Most existing systems rely on a centralized system for collecting device data. These systems lack flexibility because data is extracted in a predefined way. Also, they do not scale to a large number of devices because large volumes of raw data are transferred. In our new concept of a device database system, distributed query execution techniques are applied to leverage the computing capabilities of devices, and to reduce communication. In this article, we define an abstraction that allows us to represent a device network as a database and we describe how distributed query processing techniques are applied in this new context.

Querying the Physical World

PHILIPPE BONNET, JOHANNES GEHRKE, AND PRAVEEN SESHADRI, CORNELL UNIVERSITY

The widespread deployment of sensors, actuators, and mobile devices is transforming the physical world into a computing platform. We will soon find computing power, memory, and communication capabilities on temperature sensors and motion detectors, on door locks, light bulbs, and alarms, on every cellular phone, in every vehicle, and soon in every person's wallet or on their key ring. Emerging networking techniques ensure that devices are interconnected and accessible from local- or wide-area networks [1].

Using this new computing platform, users interact with portions of the physical world. In a large class of applications, users monitor phenomena in a given environment. Examples of monitoring applications include gathering information in a disaster area, supervising items in a factory warehouse, or controlling vehicle traffic in a large city [2, 3].

Let us take the concrete example of an existing flood detection system. For about twenty years now, the ALERT system has been deployed in several US states (http://www.alertsystems.org). A typical ALERT installation consists of several types of sensors in the field: rainfall sensors, water level sensors, weather sensors, etc. A predefined set of data is regularly extracted from each sensor, transferred to a central site and stored in a database system. Users query the database system through a graphical user interface. Here are some example queries that users can express: "For each rainfall sensor, display the average level of rainfall for 1999," Display the current level of rainfall for all sensors in Tompkins County, or "Every hour, display the location of the sensors where the level of rainfall is greater than 250 mm."

Query Processing over Device Networks

The example of the flood detection system emphasizes that monitoring is best described in a declarative manner: users submit queries concerning a device network regardless of its physical structure or its organization. In monitoring applications, users typically ask three kinds of queries:

- **Historical queries**: These are typically aggregate queries over historical data obtained from the device network, e.g., "For each rainfall sensor, display the average level of rainfall for 1999."
- **Snapshot queries**: These queries concern the device network at a given point in time, e.g., "Retrieve the current rainfall level for all sensors in Tompkins County."
- **Long-running queries**: These queries concern the device network over a time interval, e.g., "For the next five hours, retrieve every 30 seconds the rainfall level for all sensors in Tompkins County."

The existing ALERT system implements a *warehousing* approach, where data are extracted from the devices in a predefined way and stored in a centralized database system that is responsible for query processing. This warehousing approach is well suited for aggregate queries asked over historical data; however, it has two major limitations:

- **The warehousing approach dissociates access to devices from the query workload**. For instance, in an emergency situation, a fire department might require that specific data be accessed in a group of sites in order to decide on actions to take: "Every minute, display the rainfall level obtained from all sensors in Tompkins County." This long-running query cannot be answered in a traditional system if data is extracted from the sensors too infrequently. One solution would be to continuously extract all data from each device and transfer them to the database server. This solution is not feasible in practice because it might not be possible to extract all data from a sensor (e.g., a camera takes measurements in only one direction and it is not possible to measure data in all directions simultaneously) or because it is too expensive to transmit a continuous flow of raw data through the device network.
- **The warehousing approach uses valuable resources to transfer large amounts of raw data from devices to the database server**. Excessive resources are consumed at each device and on the network when transmitting large volumes of data. First, it is in general not necessary to extract data from the whole device network to answer a given query. In our example, the group of sensors sending data back to the database server should be reduced to sensors located in Tompkins County. Second, modern devices include processing capabilities that could be used to process data locally and thus reduce data transfer and energy consumption.

Our alternative to a warehousing approach is a *distributed* approach where the query workload determines the data that are extracted from remote sites, and where possibly portions

Praveen Seshradi is currently on leave at Microsoft.

 1070-9916/00/$10.00 © 2000 IEEE

of queries are executed on devices. This approach allows the database system to control the resources that are used; it is primarily targeted at snapshot and long-running queries. In addition, aggregate queries over historical data could be evaluated against materialized data stored on some devices instead of a centralized server. We call a database system that enables distributed query processing over a device network a *device database system*. We study such systems in the COUGAR Device Database Project at Cornell University.

The DataSpace project at Rutgers (http://www.cs.rutgers.edu/dataman/) recognized the advantages of the distributed approach over the warehousing approach for querying device networks [4]. In a DataSpace, devices encapsulating data can be queried, monitored, and controlled. Network primitives are developed to guarantee that only relevant devices are contacted when a query is evaluated.

Device Database Systems

In this article, we explain our new concept of a device database system, an area that we consider a very fruitful direction for new research. We will describe database abstractions for representing devices and we illustrate how queries are formulated in SQL with minimal additions to the language. Later, we use an example to show how distributed query processing techniques are applied in the new context of a device database system. We use an analytical model to illustrate the benefits of our approach.

We would like to point out that the methods described in this article represent the first generation of our system [5]. The core components of the first-generation COUGAR system are implemented and fully functional. We demonstrated the system at the Intel Computing Continuum Conference [6]. Note that in this article we do not address several of the specific research challenges that lie ahead, such as new query processing strategies to leverage computing capabilities on the devices, query processing strategies that adapt to changing conditions in the network, decentralized meta-data management, and administration. We overview these issues as we conclude.

Device Database Systems

We call a physical object with computing and communication capabilities a *device*. Some devices embed computing and communication capabilities (e.g., WINS sensor nodes [7], Smart Dust Motes [8], cell phones, or Smartcards) while others are composed of a physical object connected to an external computer (e.g., a door actuator connected to a desktop computer). Devices are interconnected and accessible from a local- or wide-area network. Some devices are stationary, others are mobile; some devices are always connected to the network, others intermittently. In this article we concentrate on stationary devices.

Database Abstractions for Representing Devices

In the warehousing approach, discussed earlier, devices are not part of the database system; they are accessed using a predefined extraction procedure that populates relations in the centralized database system. Our goal in a device database system is to access devices directly when processing queries. We thus need to represent devices in the database system.

Let us first refine our definition of devices. We consider that each device is a mini-server that supports a set of functions and can process portions of the queries directly at the device.[1] A function either (a) acquires, stores and processes data or (b) triggers an action in the physical world. Both kinds of functions return results (at least a status report or an error message). We distinguish between synchronous and asynchronous functions. Synchronous functions return results immediately, on-demand; they are used to monitor continuous phenomena, e.g., a function that returns the rainfall level. Asynchronous functions return results after an arbitrary period of time; they are used to monitor threshold events, e.g., a function that detects an abnormal rainfall level. Functions provided by an intermittently connected device can only return results when the device is connected; they are asynchronous functions. Stationary devices, e.g., rainfall sensors, may support both synchronous and asynchronous functions.

We need to represent the set of functions provided by devices at the database level. We distinguish two levels of representation:

- User representation — how are devices modeled in the database schema?
- Internal representation — how are devices represented internally?

User Representation — Today's object-relational and object-oriented databases support Abstract Data Type (ADT) objects that are single-attribute values encapsulating a collection of related data [9]. Note that there are natural parallels between devices and ADTs. Both ADTs and devices provide controlled access to encapsulated data through a well defined interface. We build upon this observation by modeling each type of device in the network as an ADT. The public interface of the ADT corresponds to the specific functions supported by the device. An actual ADT object in the database corresponds to a physical device in the real world.

Let us model the database schema corresponding to the flood detection example earlier. We consider a simplified schema that consists of the following relation:

RFSensors(Sensor, X, Y)

A record in the *RFSensors* relation has three attributes. The first attribute, called *Sensor*, is an ADT that represents the physical rainfall sensors. The actual sensor data is located on the rainfall sensor; the ADT *Sensor* provides functions for accessing the data. For example, *Sensor.getRainfallLevel()* returns the current level of rainfall measured in mm. The other two attributes denote the location of the sensor according to some coordinate system.

Internal Representation — Before discussing the internal representation of ADT functions, let us recall some background knowledge about query processing and the internals of a database system. Query processing classically proceeds as follows. The database system accepts a query, produces a query execution plan, executes this plan against the database, and produces the answer. The execution plan is the internal blueprint for evaluating a query. It combines algebraic operators (e.g., selection, projection, and join operators in the relational algebra), which serve as the basic building blocks for manipulating data (i.e., relations that are sets of records).

In object-relational database systems, ADT functions are either represented as expressions [9] or as joins involving virtual relations [10].[2] When an expression containing an ADT function is evaluated, a (local) function is called to obtain its

[1] *Embedding a database server on a device is realistic. All major database vendors propose database servers for palm-sized PCs, which represent the processing capabilities that we can expect from all devices in a near future.*

[2] *Table functions defined in IBM DB2 associate a user-defined function with a virtual relation.*

return value. It is assumed that this return value is readily available on-demand. This assumption does not hold in a device database system for two reasons. First, functions corresponding to device ADT functions may incur high latency due to their distant location from the database server. Second, some device functions are asynchronous and thus a call to such a function may incur an arbitrary delay.

A *virtual relation* is a tabular representation of a function. A record in a virtual relation (called a virtual record) contains the input arguments and the output argument of the function with which it is associated.[3] Such relations are called virtual because they are not actually defined in the database schema, as opposed to *base relations*. In COUGAR, we use virtual relations for the internal representation of device functions.

If a device function M takes m arguments, then the schema of its associated virtual relation Attrs(VR) has $m+3$ attributes, where the first attribute corresponds to the unique identifier of a device (i.e., the identifier of an actual device ADT object), attributes 2 to $m+1$ correspond to the input arguments of M, attribute $m+2$ corresponds to the output value of M, and attribute $m+3$ is a time stamp corresponding to the point in time at which the output value is obtained.[4] We assume global time. Each time stamp thus determines a position in an ordered domain shared across all devices. As a consequence, each virtual relation could be considered as a sequence [11].

In our example, the database schema consists of one base relation (*RFSensors*) and of a virtual relation *VRFSensorsGetRainfallLevel* for the function *getRainfallLevel()*. Since this function takes no input arguments, the virtual relation has three attributes: *Sensor*, *Level*, and *TimeStamp*, i.e., the identifier of the Sensor device, the *Level* of rainfall measured, and the associated *TimeStamp*.

Note that a virtual relation has specific properties:

- A virtual relation is append-only; new records are inserted in a virtual relation when the associated device function returns a result. Records in a virtual relation are never updated or deleted.
- A virtual relation is naturally partitioned across all devices represented by the same device ADT. Each device function contributes to a portion of the virtual relation to which it is associated.

The latter observation has an interesting consequence: **a collection of devices is internally represented as a distributed database**. Virtual relations are partitioned across a set of devices. Base relations are either stored on a central database server or partitioned across devices.[5]

The Cougar System consists of a front-end server connected to a set of devices. The front-end includes a full-fledged database server. Devices include a lightweight query execution engine that is responsible for accessing virtual relations and for processing query fragments that involve these virtual relations.

Queries over a Device Database

Recall that we consider historical queries, snapshot queries, and long-running queries over a device network. Historical and snapshot queries are naturally formulated as declarative queries in SQL. Long-running queries are also formulated in SQL with little modifications to the language. We add clauses for specifying the duration of a long-running query; the choice of syntax is arbitrary.

Because of space limitation, we do not describe the complete query semantics here; the interested reader is referred to Bonnet *et al*. [12] for details. Note that long-running queries involving time windows (in particular aggregates over time windows) are best expressed using temporal extensions to the relational model [13, 14] or using a sequence model [11].

We now provide an example of a long-running query based on the flood detection application presented earlier.

Query Q: "Retrieve every 30 seconds the rainfall level if it is greater than 50 mm."

```
SELECT R.Sensor.getRainfallLevel()
FROM RFSensors R
WHERE R.Sensor.getRainfallLevel() > 50
 AND $every(30);
```

The function *$every(30)* specifies that a new record is inserted every 30 seconds into the append-only virtual relation corresponding to the function *RFSensor.getRainfallLevel()*. This record is propagated within the query execution plan chosen for the long-running query, and possibly a new answer is generated. Note that a long-running query is not evaluated by repeatedly executing the declarative query over the new records inserted in the virtual relations. (This would be a form of polling and it would introduce an arbitrary delay into the processing of device data.)

Query Processing in a Device Database System

In this section, we concentrate on a simple example to give an overview of query processing and to show the benefits of the distributed query processing approach versus a warehousing approach. Because of space limitation, we do not cover here all the issues related to query processing in a device database system. We first define new performance metrics and then discuss our example.

Performance Metrics

When processing a query, a database system first constructs an execution plan. The query optimizer is responsible for generating the execution plan that minimizes a given cost function.

The traditional performance metrics in a database system are throughput and response time. Throughput is the average number of queries processed per unit of time; it depends on the total work performed in the system to evaluate a query. Response time is the time needed by the system to produce all answer records to a query.

For long-running queries in a device database system, the traditional performance metric of query response time becomes obsolete: the query will always run for a given time interval, with varying resource usage.

We define two new metrics that correspond to the performance goals of a device database system:

- **Resource usage**: The total amount of energy consumed by the devices when executing a query. Resource usage is expressed in Joules.
- **Reaction time**: The interval between the time a function, called on a device, returns a value, and the time the corresponding answer is produced on the front-end. Reaction time is expressed in seconds.

The problem now is twofold:

- To define cost models for resource usage and reaction time.

[3] *We assume without loss of generality that a device function has exactly one return value; an extension to the general case is straightforward.*

[4] *Note that for mobile devices, we might integrate the location of the device as an additional attribute in the virtual relation.*

[5] *It is particularly interesting to partition a base relation that references a device ADT in a system where devices frequently join or leave the network; partitioning the base relation thus avoids maintaining centralized information concerning the devices currently in the system.*

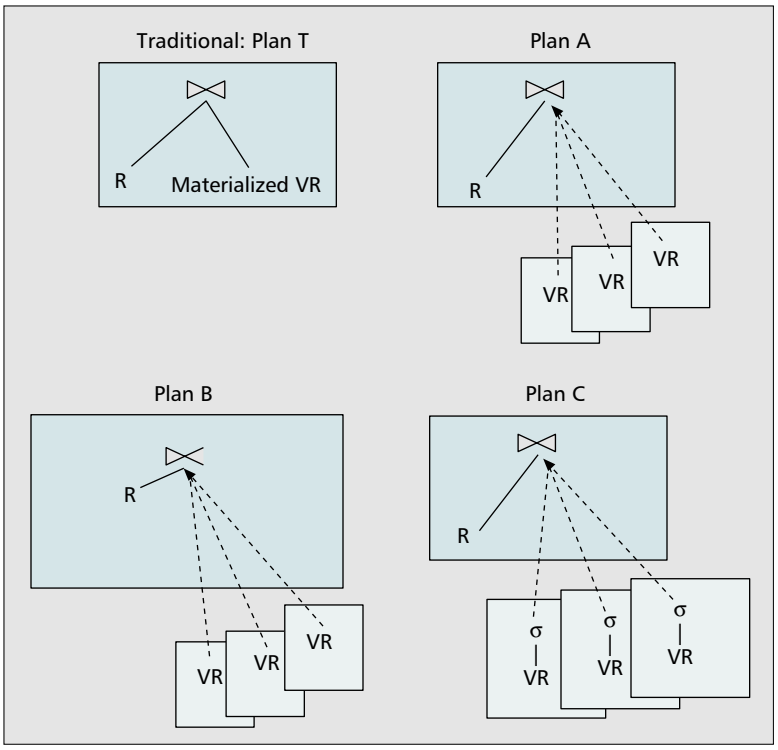

Figure 1. *Execution plans for Query Q1.*

- To obtain and maintain correct settings for the system parameters from the cost model, i.e., settings that actually reflect the status of a given device database system over time.

Example

Our goal in this section is not to cover all issues related to query processing in a device database system, but rather to illustrate how existing distributed database techniques can be applied in this new context [15, 16]. We discuss the characteristics of device database systems with respect to existing distributed database systems and use an analytical model to illustrate the benefits of our approach.

Query Q1: "Retrieve every 30 seconds the rainfall level if it is greater than 50 mm."

```
SELECT VR.value
FROM VRFSensorsGetRainfallLevel VR, RFSensors R
WHERE VR.Sensor = R.Sensor AND VR.value > 50
      AND $every(30);
```

We use as our example the query Q1, which is the result of rewriting query Q using the virtual relation *VRFSensorsGetRainfallLevel*. This query could be used to monitor the evolution of rainfall in flooded areas. We consider a system with 200 devices; the cardinality of relation *R* is therefore 200 records. Query Q1 is run as a long-running query with a duration of four hours. The rainfall level is measured every 30 seconds; as a result, up to 480 virtual records are inserted into each partition of the virtual relation.

Distributed Query Execution Plans — SQL queries usually have a large space of possible execution plans. These are obtained by considering various shapes for the tree of relational operators, by permuting the position of relational operators in this tree, by choosing various implementations for a relational operator (in particular, each database system implements a set of join methods, e.g., nested loop, sort-merge, hash-join, semi-join), and by permuting the relative position of sub-trees [17]. In a distributed context, the execution plan reflects the distributed nature of the database: it is composed of query fragments, i.e., sub-trees of relational operators, assigned to execution sites. Three more dimensions are thus added to the space of possible execution plans: What are the candidate execution sites? How are query fragments associated to execution sites? What is the strategy for transferring data from one site to another?

Figure 1 presents four execution plans for Q1; each plan is a tree of relational operators that manipulate base and virtual relations. Plan T represents the execution plan that would be generated for Query Q1 in a traditional system such as ALERT. Data extracted from the devices are materialized in the relation VR that is located on the front-end (represented as a darker shaded rectangle). The execution plan is a simple tree composed of one join operator between relation R and relation VR (using joining condition *R.Sensor = VR.Sensor AND VR.value > 50)*. This join is executed on the front end.

The other execution plans illustrate the use of distributed database techniques in a device database system. Plan A is also a simple tree where R is joined on the front end with relation VR partitioned across a set of devices (represented as lighter shaded rectangles). This execution plan is evaluated as follows. The front end asks each device to measure rainfall level and to transfer the resulting virtual records back to the front end. (Virtual records are produced once on each site for a snapshot query, and repetitively for a long-running query). Each virtual record arriving on the front end is then joined with relation R.

Intuitively, this execution plan is not optimal: all devices with rainfall sensors transmit data to the front end while the query only concerns the sensors that measure a rainfall level greater than 50. An alternative execution plan pushes the join to the devices, thus trading increased processing on devices for reduced network traffic. Instead of pushing the join between R and VR to each device, Plan B defines a semi-join between relation R and the partitions of the virtual relation VR located on the devices [16]. The semi-join projects out the joining attribute from relation R (here the device id *Sensor*) and sends the resulting relation to all devices; a semi-join avoids transferring the complete relation R to all devices. On the devices, whenever the rainfall level is measured, a virtual record is generated and it is joined with the portion of relation R sent by the front end (using joining condition *R.Sensor = VR.Sensor and VR.value > 50*). If the joining condition is verified, then the virtual record is sent back to the front end, where it is joined with complete records from relation R (not only the joining attribute). Only the sensors whose rainfall level is greater than 50 send data back to the front end.

A third execution plan only pushes the selection (*VR.value > 50*) onto the devices; only records that verify this condition are sent back to the front end, where they are joined with relation R. Plan C represents this execution plan. Compared to Plan B, there is no subset of relation R transmitted to the devices. We compare the resource usage of these three execution plans in the next section.

Analytical Model — We use a simple analytical model to compare the costs of the three execution plans identified in the previous section. We assume a multi-cluster, single-hop WINS network [7]. There are 20 clusters each containing 10 devices. We consider the total energy consumed per sensor node as the linear combination of CPU costs, the cost of a memory access, the cost of sending a message, and the cost of sending *N* bytes on the network:

Cost in Joules = Wcpu * CPU + Wram * RAM
+ Wsend * NbMsgs + Wbdw * SizeMsgs

The weight factors are used to transfer all components of the cost into Joules. Table 1 lists the weight factors we used for our experiments. The factors were obtained by W. Kaiser and G. Pottie, through measurements in a WINS network composed of sensor nodes from Sensoria Corp. [7]. The energy needed by the processor to operate dominates the energy needed by the RAM, so we set Wram = 0. The cost per record of a join or a selection is NbInstPerComp instructions. We do not model the cost of invoking the device function. The cost per message is due to synchronization between the sending and receiving nodes. We consider that nodes are 30 meters from each other. In this case the cost of sending 1000 bytes is 0.23J. (Note that the capacity of a battery on a WINS sensor node is 3.5E4 Joules.) We further assume that the size of each virtual record is 50 bytes.

Wcpu	0.000001 J/instruction
Wram	0
Wsend	0.059 J/msg
Wbdw	0.23 J/Kbytes
NbInstPerComp	5000

Table 1. *Parameters and settings for modeling resource usage.*

We study resource usage on sensor nodes directly involved in the query (i.e., the nodes on which a partition of the virtual relation is located); we do not consider resource usage on the nodes that are traversed for communication purposes. Each sensor node satisfies the condition in query Q1 (*Vr.value > 50*) with a certain probability. We trace the resource usage in the two extreme cases, i.e., for sensor nodes that are always located outside a flood area and whose rainfall level is thus never greater than 50, and for sensor nodes that are always located inside a flood area.

Figure 2 traces the resource usage expressed in Joules as a function of time (given that the rainfall level is measured every 30 seconds) for nodes always located outside a flood area. With Plan A, data is sent back to the front end whenever it is generated. With Plan B and, respectively, Plan C, a join and, respectively, a selection, are pushed to the device. As a result, the condition on the rainfall level is checked on the devices and none of the devices located outside a flood area sends data back to the front end. Plan B pays the initial cost of transferring a fragment of relation R to the devices. This initial cost is amortized (compared to Plan A) during the lifespan of a long-running query.

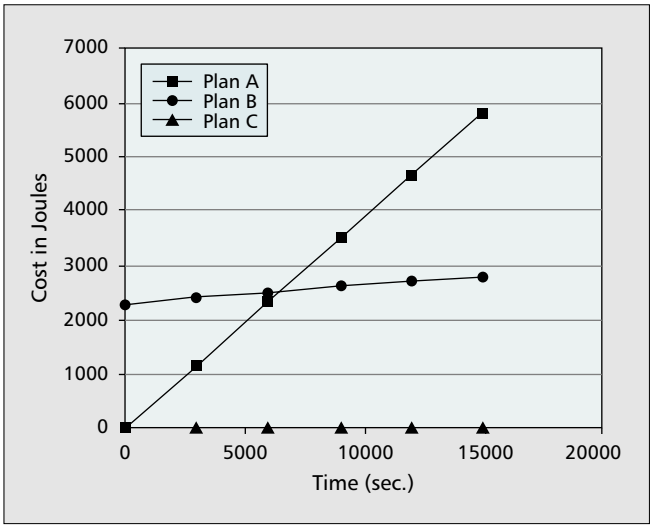

Figure 2. *Resource usage for sensors located outside a flood area.*

Figure 3 traces the resource usage expressed in Joules as a function of time (given that the rainfall level is measured every 30 seconds) for nodes always located inside a flood area. With all plans, data is always sent back to the front end. The initial cost of Plan B is here never amortized. Plan C and Plan A have almost similar curves; this illustrates that the cost of performing a selection is low compared to the cost of sending data.

In this example, pushing a selection as in Plan C is the optimal choice. This is intuitive since the query filters out uninteresting events generated on the devices. Pushing the selection allows the device database system to trade efficiently increased processing on the devices for reduced communication.

Conclusions

In the near future, devices with processing and communication capabilities will be deployed in the physical world, providing a powerful computing platform. The first generation of the Cornell Cougar systems demonstrates that the application of database technology to this new computing platform shows much promise for providing flexible and scalable access to large collections of devices. Our work has introduced a set of research problems, and we now provide a brief overview of some of the questions that our ongoing research is addressing:

Meta-data management: Current distributed database optimizers assume global knowledge, i.e., the optimizer has access to exact meta-information about the complete system. In a device database system, we cannot assume that a single site maintains global knowledge about the system because of the large scale and dynamic nature of a device network, and because it would incur a significant administration overhead. How can we maintain meta-data in a decentralized way and how can we utilize this information to devise good query plans?

Query processing: Query processing should take advantage of the computing capabilities at the devices in order to minimize the total amount of resources consumed in the device network while minimizing reaction time. In addition, conditions in a device network change over time. Devices fail, move, or disconnect, the network topology may evolve, and batteries are used and recharged. Thus query plans must adapt dynamically

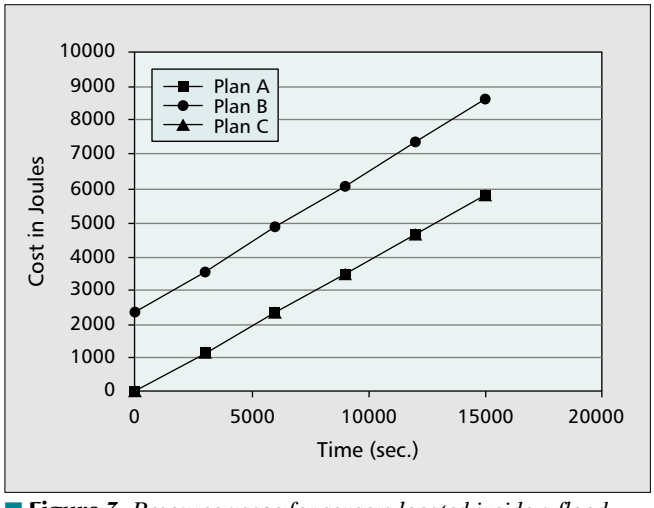

Figure 3. *Resource usage for sensors located inside a flood area.*

to changing network conditions and they must show a certain degree of robustness against device failures. In addition, for long-running queries the conditions in the device network might change significantly while the query runs.

Acknowledgments

We thank Stephane Bressan and Tobias Mayr, who helped debug earlier versions of this article, as well as the reviewers for helpful comments. This article benefited from interactions with the SensIT community. In particular, Bill Kaiser provided valuable information concerning the Sensoria WINS network. This work is sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Material Command, USAF, under agreement number F-30602-99-0528, by the National Science Foundation under Grant No. EIA 97-03470, by NSF Grant IIS-9812020, and by a grant from Microsoft Research to Philippe Bonnet.

References

[1] D. Estrin, R. Govindan, and J. Heidemann (Eds.), "Embedding the Internet," *Communications of the ACM*, vol. 43, no. 5. May 2000.
[2] DARPA: SenseIT Project, http://www.darpa.mil/ito/research/sensit/background.html
[3] D. Estrin *et al.*, "Next Century Challenges: Scalable Coordination in Sensor Networks," *Mobicom '99*, Seattle, Washington, pp. 263–70.
[4] T. Imielinski and S. Goel, "DataSpace: Querying and Monitoring Deeply Networked Collections in Physical Space," MobiDE 1999, pp. 44–51.
[5] P. Bonnet and P. Seshadri, "Device Database Systems, poster paper," *Proc. Int'l. Conf. Data Engineering ICDE'99*, San Diego, CA, Mar. 2000.
[6] *Intel Computing Continuum Conf.*, http://www.intel.com/intel/cccon/
[7] J. M. Kahn, R. H. Katz, and K. S. J. Pister, "Mobile Networking for Smart Dust," *ACM/IEEE Intl. Conf. Mobile Computing and Networking (MobiCom '99)*, Seattle, WA, Aug. 17–19, 1999
[8] G. Pottie and W. Kaiser, "Wireless Integrated Network Sensors (WINS): Principles and Approach," *Communications of the ACM*, vol. 43, no. 5, May 2000.
[9] P. Seshadri, "Enhanced Abstract Data Types in Object-Relational Databases," *VLDB Journal*, vol. 7, no. 3, 1998, pp. 130–40.
[10] U. Schreierf *et al.*, "Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS," *VLDB*, 1991, pp. 469–78.
[11] P. Seshadri, M. Livny, and R. Ramakrishnan, "SEQ: A Model for Sequence Databases," *ICDE* 1995.
[12] P. Bonnet *et al.*, "Query Processing in a Device Database System," Cornell Technical Report TR99-1775, Oct. 1999.
[13] A. Tansel *et al.*, *Temporal Database: Theory, Design and Implementation*, Benjamin/Cummings, 1993.
[14] A.Dekhtyar, R. Ross, and V. S. Subrahmanian, "Probabilistic Temporal Databases: Algebra," Jan. 1999, University of Maryland technical report CS-TR-3987, submitted to *ACM Trans. Database Systems*.
[15] L. F. Mackert and G. M.Lohman, "R* Optimizer Validation and Performance Evaluation for Distributed Queries," *Proc. Int'l. VLDB Conf.*, pp. 149–59, Kyoto, Japan, Aug. 1986, Management Systems. ICOD 1980: pp. 204–15.
[16] C. T. Yu, "Distributed Database Query Processing," *Query Processing in Database Systems*, 1985: pp. 48–61.
[17] R. Ramakrishnan and J. Gehrke, *Database Management Systems, Second Edition*, McGraw Hill, 1999.

Biographies

PHILIPPE BONNET (bonnet@cs.cornell.edu) received a Ph.D. from the Universite de Savoie in 1999. He is currently a research associate in the Department of Computer Science at Cornell University. His current research interests involve device database systems, database tuning, and next-generation database systems. He is a member of the ACM and the IEEE Computer Society.

JOHANNES GEHRKE (johannes@cs.cornell.edu) received his Ph.D. degree from the University of Wisconsin in 1999. He is currently an assistant professor in the Department of Computer Science at Cornell University. His research is in the areas of data mining and database systems. He is the recipient of an IBM Faculty Award and he serves on the editorial board of *Knowledge and Information Systems*. He is the co-author of the textbook *Database Management Systems (Second Edition)* published by McGraw Hill in 1999. He is a member of the ACM and the IEEE Computer Society.

PRAVEEN SESHADRI (praveen@cs.cornell.edu) received a Ph.D. from the University of Wisconsin-Madison in 1996. He is an assistant professor in the Department of Computer Science at Cornell University, currently on leave at Microsoft. His research is in the area of next-generation database systems and data management for personal devices. He received an IBM Faculty Award and an NSF Career Award. He is a member of the ACM and the IEEE Computer Society.

Appendix B:

Towards Sensor Database Systems

Philippe Bonnet, Johannes Gehrke, Praveen Seshadri[1]

Computer Science Department, Upson Hall
Cornell University
Ithaca, NY, 14853 USA
{bonnet,johannes,praveen}@cs.cornell.edu

Abstract. Sensor networks are being widely deployed for measurement, detection and surveillance applications. In these new applications, users issue long-running queries over a combination of stored data and sensor data. Most existing applications rely on a centralized system for collecting sensor data. These systems lack flexibility because data is extracted in a predefined way; also, they do not scale to a large number of devices because large volumes of raw data are transferred regardless of the queries that are submitted. In our new concept of sensor database system, queries dictate which data is extracted from the sensors. In this paper, we define a model for sensor databases. Stored data are represented as relations while sensor data are represented as time series. Each long-running query formulated over a sensor database defines a persistent view, which is maintained during a given time interval. We also describe the design and implementation of the COUGAR sensor database system.

1 Introduction

The widespread deployment of sensors is transforming the physical world into a computing platform. Modern sensors not only respond to physical signals to produce data, they also embed computing and communication capabilities. They are thus able to store, process locally and transfer the data they produce. Still, at the heart of each sensor, a set of signal processing functions transform physical signals such as heat, light, sound, pressure, magnetism, or a particular motion into sensor data, i.e., measurements of physical phenomena as well as detection, classification or tracking of physical objects.

Applications monitor the physical world by querying and analyzing sensor data. Examples of monitoring applications include supervising items in a factory warehouse, gathering information in a disaster area, or organizing vehicle traffic in a large city [6]. Typically, these applications involve a combination of stored data (a list of sensors and their related attributes, such as their location) and sensor data. We call these *sensor databases*. This paper focuses on sensor query processing – the design, algorithms, and implementations used to run queries over sensor databases. The

[1] Praveen Seshadri is currently on leave at Microsoft: 3/1102 Microsoft, One Microsoft Way, Redmond, WA. pravse@microsoft.com.

concepts developed in this paper were developed under the DARPA Sensor Information Technology (SensIT) program [22].

We define a *sensor query* as a query expressed over a sensor database. A typical monitoring scenario involves aggregate queries or correlation queries that give a bird's eye view of the environment as well as queries zooming on a particular region of interest. Representative sensor queries are given below in Example 1.

Example 1 (Factory Warehouse): Each item of a factory warehouse has a stick-on temperature sensor attached to it. Sensors are also attached to walls and embedded in floors and ceilings. Each sensor provides two signal-processing functions: (a) *getTemperature()* returns the measured temperature at regular intervals, and (b) *detectAlarmTemperature(threshold)* returns the temperature whenever it crosses a certain threshold. Each sensor is able to communicate this data and/or to store it locally. The sensor database stores the identifier of all sensors in the warehouse together with their location and is connected to the sensor network. The warehouse manager uses the sensor database to make sure that items do not overheat. Typical queries that are run continuously include:

- Query 1: "Return repeatedly the abnormal temperatures measured by all sensors."
- Query 2: "Every minute, return the temperature measured by all sensors on the third floor".
- Query 3: "Generate a notification whenever two sensors within 5 yards of each other simultaneously measure an abnormal temperature".
- Query 4: "Every five minutes retrieve the maximum temperature measured over the last five minutes".
- Query 5: "Return the average temperature measured on each floor over the last 10 minutes".

These example queries have the following characteristics:

- Monitoring queries are long running.
- The desired result of a query is typically a series of notifications of system activity (periodic or triggered by special situations).
- Queries need to correlate data produced simultaneously by different sensors.
- Queries need to aggregate sensor data over time windows.
- Most queries contain some condition restricting the set of sensors that are involved (usually geographical conditions).

As in relational databases, queries are easiest to express at the logical level. Queries are formulated regardless of the physical structure or the organization of the sensor network. The actual structure and population of a sensor network may vary over the lifespan of a query.

Clearly, there are similarities with relational database query processing. Indeed, most applications combine sensor data with stored data. However, the features of sensor queries described here do not lend themselves to easy mapping to relational databases and sensor data is different from traditional relational data (since it is not stored in a database server and it varies over time).

There are two approaches for processing sensor queries: the warehousing approach and the distributed approach. The warehousing approach represents the current state-of-the-art. In the warehousing approach, processing of sensor queries and access to the sensor network are separated. (The sensor network is simply used by a data

collection mechanism.) The warehousing approach proceeds in two steps. First, data is extracted from the sensor network in a predefined way and is stored in a database located on a unique front-end server. Subsequently, query processing takes place on the centralized database. The warehousing approach is well suited for answering predefined queries over historical data.

The distributed approach has been described in [2][3] and is the focus of this paper. In the distributed approach, the query workload determines the data that should be extracted from sensors. The distributed approach is thus flexible – different queries extract different data from the sensor network – and efficient – only relevant data are extracted from the sensor network. In addition, the distributed approach allows the sensor database system to leverage the computing resources on the sensor nodes: a sensor query can be evaluated at the front-end server, in the sensor network, at the sensors, or at some combination of the three.

In this paper, we describe the design space for a sensor database system and present the choices we have made in the implementation of the Cornell COUGAR system. This paper makes the following contributions:

1. We build on the results of Seshadri et al. [19] to define a data model and long-running queries semantics for sensor databases. A sensor database mixes stored data and sensor data. Stored data are represented as relations while sensor data are represented as time series. Each long-running query defines a persistent view, which is maintained during a given time interval.
2. We describe the design and implementation of the Cornell COUGAR sensor database system. COUGAR extends the Cornell PREDATOR object-relational database system. In COUGAR, each type of sensor is modeled as a new Abstract Data Type (ADT). Signal-processing functions are modeled as ADT functions that return sensor data. Long-running queries are formulated in SQL with little modifications to the language. To support the evaluation of long-running queries, we extended the query execution engine with a new mechanism for the execution of sensor ADT functions. The initial version of this system has been demonstrated at the Intel Computing Continuum Conference [7].

Addressing these issues is a necessary first step towards a sensor database system. In addition, a sensor database system should account for sensor and communication failures; it should consider sensor data as measurements with an associated uncertainty not as facts; finally, it should be able to establish and run a distributed query execution plan without assuming global knowledge of the sensor network. We believe that these challenging issues can only be addressed once the data model and internal representation issues have been solved.

2 A Model for Sensor Database Systems

In this section, we introduce definitions for sensor databases and sensor queries. We build on existing work by Seshadri et al [19] to define a data model for sensor data and an algebra of operators to formulate sensor queries.

2.1 Sensor Data

A sensor database involves stored data and sensor data. Stored data include the set of sensors that participate in the sensor database together with characteristics of the sensors (e.g., their location) or characteristics of the physical environment. These stored data are best represented as relations. The question is: how to represent sensor data? First, sensor data are generated by signal processing functions. Second, the representation we choose for sensor data should facilitate the formulation of sensor queries (data collection, correlation in time, and aggregates over time windows).

Note that time plays a central role. Possibly, signal processing functions return output repeatedly over time, and each output has a time-stamp. In addition, monitoring queries introduce constraints on the sensor data time-stamps, e.g., Query 3 in Example 1 assumes that the abnormal temperatures are detected either simultaneously or within a certain time interval. Aggregates over time windows, such as Query 4 and 5, reference time explicitly.

Given these constraints, we represent sensor data as time series. Our representation of sensor time series is based on the sequence model introduced by Seshadri et al. [19]. Informally, a sequence is defined as a 3-tuple containing a set of records R, a countable totally ordered domain O (called ordering domain – the elements of the ordering domain are referred to as positions) and an ordering of R by O. An ordering of a set of records R by an ordering domain O is defined as a relation between O and R, so that every record in R is associated with some position in O. Sequence operators are n-ary mappings on sequences; they operate on a given number of input sequences producing a unique output sequence. All sequence operators can be composed. Sequence operators include: select, project, compose (natural join on the position), and aggregates over a set of positions. Because of space limitation, we refer the reader to [4] for a formal definition of sensor time series

We represent sensor data as a time series with the following properties:

1. The set of records corresponds to the outputs of a signal processing function over time.
2. The ordering domain is a discrete time scale, i.e. a set of time quantum; to each time quantum corresponds a position. In the rest of the paper, we use natural numbers as the time-series ordering domain. Each natural number represents the number of time units elapsed between a given origin and any (discrete) point in time. We assume that clocks are synchronized and thus all sensors share the same time scale.
3. All outputs of the signal processing function that are generated during a time quantum are associated to the same position p. Note that, in case a sensor does not generate events during the time quantum associated to a position, the Null record is associated to that position.
4. Whenever a signal processing function produces an output, the base sequence is updated at the position corresponding to the production time. Updates to sensor time series thus occur in increasing position order.

2.2 Sensor Queries

Sensor queries involve stored data and sensor data, i.e. relations and sequences. We define a sensor query as an acyclic graph of relational and sequence operators. The inputs of a relational operator are base relations or the output of another relational operator; the inputs of a sequence operator are base sequences or the output of another sequence operator, i.e. relations are manipulated using relational operators and sequences are manipulated using sequence operators. There are three exceptions to this rule. Three operators allow combining relations and sequences: (a) the relational projection operator can take a sequence as input and project out the position attribute to obtain a relation, (b) a cross product operator can take as input a relation and a sequence to produce a sequence and (c) an aggregate operator can take a sequence as input and a grouping list that does not include the position attribute.

Sensor queries are long running. To each sensor query is associated a time interval of the form [O, O + T] where *O* is the time at which it is submitted and *T* is the number of time quantums (possibly 0) during which it is running.

During the span of a long-running query, relations and sensor sequences might be updated. An update to a relation R can be an insert, a delete, or modifications of a record in R. An update to a sensor sequence S is the insertion of a new record associated to a position greater than or equal to all the undefined positions in S (see Section 3.1.1). Concretely, each sensor inserts incrementally the set of records produced by a signal processing function at the position corresponding to the production time.

A sensor query defines a view that is persistent during its associated time interval. This persistent view is maintained to reflect the updates on the sensor database. In particular, the view is maintained to reflect the updates that are repeatedly performed on sensor time series.

Jagadish et al. [13] showed that persistent views over relations and sequences could be maintained incrementally without accessing the complete sequences, given restrictions on the updates that are permitted on relations and sequences, and given restrictions on the algebra used to compose queries. Informally, persistent views can be maintained incrementally if updates occur in increasing position order and if the algebra used to compose queries does not allow sequences to be combined using any relational operators. Both conditions hold in our definition of a sensor database.

3 The COUGAR Sensor Database System

In this section, we discuss the representation of sensor data, as well as the formulation and evaluation of sensor queries in the initial version of COUGAR. We discuss the limitations of this system and the conclusions that we have drawn.

We have introduced in Section 2 a model of sensor database. We took a set of design decisions when implementing this model in the COUGAR system. We distinguish between the decisions we took concerning:

1. User representation: How are sensors and signal processing functions modeled in the database schema? How are queries formulated?

2. Internal representation: How is sensor data represented within the database components that perform query processing? How are sensor queries evaluated to provide the semantics of long-running queries?

3.1 User Representation

In COUGAR, signal-processing functions are represented as Abstract Data Type (ADT) functions. Today's object-relational databases support Abstract Data Types that provide controlled access to encapsulated data through a well-defined set of functions [20]. We define a Sensor ADT for all sensors of a same type (e.g., temperature sensors, seismic sensors). The public interface of a Sensor ADT corresponds to the specific signal-processing functions supported by a type of sensor. An ADT object in the database corresponds to a physical sensor in the real world.

Signal-processing functions are modeled as scalar functions. Repeated outputs of an active signal processing functions are not explicitly modeled as sequences but as the result of successive executions of a scalar function during the span of a long-running query. This decision induced some limitation. Indeed, as we will see below, queries containing explicit time constraints (such as aggregates over time windows) cannot be expressed.

Sensor queries are formulated in SQL with little modifications to the language. The 'FROM' clause of a sensor query includes a relation whose schema contains a sensor ADT attribute (i.e., a collection of sensors). Expressions over sensor ADTs can be included in the 'SELECT' or in the 'WHERE' clause of a sensor query.

The queries we introduced in Section 1 are formulated in COUGAR as follows. The simplified schema of the sensor database contains one relation *R(loc point, floor int, s sensorNode)*, where *loc* is a point ADT that stores the coordinates of the sensor, *floor* is the floor where the sensor is located in the data warehouse and *sensorNode* is a Sensor ADT that supports the methods *getTemp()* and *detectAlarmTemp(threshold)*, where *threshold* is the threshold temperature above which abnormal temperatures are returned. Both ADT functions return temperature represented as float.

- Query 1: "Return repeatedly the abnormal temperatures measured by all sensors"
  SELECT R.s.detectAlarmTemp(100)
  FROM R
  WHERE $every();
  The expression *$every()* is introduced as a syntactical construct to indicate that the query is long-running.
- Query 2: "Every minute, return the temperature measured by all sensors on the third floor".
  SELECT R.s.getTemp()
  FROM R
  WHERE R.floor = 3 AND $every(60);
  The expression *$every()* takes as argument the time in seconds between successive outputs of the sensor ADT functions in the query.
- Query3: "Generate a notification whenever two sensors within 5 yards of each other measure simultaneously an abnormal temperature".

SELECT R1s.detectAlarmTemp(100), R2.s. detectAlarmTemp (100)
FROM R R1, R R2
WHERE $SQRT($SQR(R1.loc.x – R2.loc.x) + $SQR(R1.loc.y – R2.loc.y)) < 5
AND R1.s > R2.s AND $every();

This formulation assumes that the system incorporates an equality condition on the time at which the temperatures are obtained from both sensors.

Query 4 and Query 5 cannot be expressed in our initial version of COUGAR because aggregates over time windows are not supported.

In COUGAR, the time interval associated with long-running queries is the interval between the instant the query is submitted and the instant the query is explicitly stopped.

3.2 Internal Representation

Query processing takes place on a database front-end while signal-processing functions are executed on the sensor nodes involved in the query. The query execution engine on the database front-end includes a mechanism for interacting with remote sensors. On each sensor a lightweight query execution engine is responsible for executing signal processing functions and sending data back to the front-end.

In COUGAR, we assume that there are no modifications to the stored data during the execution of a long-running query. Strict two-phase locking on the database front-end ensures that this assumption is verified.

The initial version of COUGAR does not consider a long-running query as a persistent view; the system only computes the incremental results that could be used to maintain such a view. These incremental results are obtained by evaluating sensor ADT functions repeatedly and by combining the outputs they produce over time with stored data.

The execution of Sensor ADT functions is the central element of sensor queries execution. In the rest of the section, we show why the traditional execution of ADT functions (which is explained below) is inappropriate for sensor queries and we present the mechanisms we have implemented in COUGAR to evaluate sensor ADT functions.

Problems with the Traditional ADT Functions Execution

In most object-relational database systems, ADT functions are used to form expressions together with constants and variables. When an expression containing an ADT function is evaluated, a (local) function is called to obtain its return value. It is assumed that this return value is readily available on-demand. This assumption does not hold in a sensor database for the following reasons:

1. Scalar sensor ADT functions incur high latency due to their location or because they are asynchronous;
2. When evaluating long-running queries, sensor ADT functions return multiple outputs.

To illustrate these problems, let us consider Query 1 in our example. One possible execution plan for Query 1 would be the following. For each temperature sensor in the relation R, the scalar function detectAlarmTemp(100) is applied.

There is a serious flaw in this execution. First, the function *detectAlarmTemp (100)* is asynchronous, i.e. it returns its output after an arbitrary amount of time. While the system is requesting an abnormal temperature on the first sensor of the relation *R*, the other temperature sensors have not been yet been contacted. It may very well be that some temperature sensors could have detected temperatures greater than 100, while the system is blocked waiting for the output of one particular function.

Second, during the span of a long-running query, *detectAlarmTemp (100)* might return multiple outputs. The evaluation plan we presented scans relation *R* once and thus does not respect the semantics of long running queries we have introduced in Section 2.

Virtual Relations

To overcome the problems outlined in the previous paragraph, we introduced a relational operator to model the execution of sensor ADT functions. This relational operator is a variant of a join between the relation that contains the sensor ADT attribute and the sensor ADT function represented in a tabular form. We call the tabular representation of a function a virtual relation.

A virtual relation is a tabular representation of a method. A record in a virtual relation (called a virtual record) contains the input arguments and the output argument of the method it is associated with[2]. Such relations are called virtual because they are not actually materialized, as opposed to base relations, which are defined in the database schema.

If a method M takes m arguments, then the schema of its associated virtual relation has m+3 attributes, where the first attribute corresponds to the unique identifier of a device (i.e., the identifier of an actual device ADT object), attributes 2 to m+1 correspond to the input arguments of M, attribute m+2 corresponds to the output value of M and attribute m+3 is a time stamp corresponding to the point in time at which the output value is obtained. In our example Query 1, the virtual relation *VRdetectAlarmTemp* is defined for the Sensor ADT function *detectAlarmTemp()*. Since this function takes one input arguments, the virtual relation has four attributes: *SensorId, Temp, Value, and TimeStamp, i.e.*, the identifier of the Sensor device that produces the data *SensorId*, the input threshold temperature *Temp*, the *Value* of the measured temperature and the associated *TimeStamp*.

We observe the following:

- A virtual relation is append-only: New records are appended to a virtual relation when the associated signal processing function returns a result. Records in a virtual relation are never updated or deleted.
- A virtual relation is naturally partitioned across all devices represented by the same sensor ADT: A virtual relation is associated to a sensor ADT function, to each sensors of these type is associated a fragment of the virtual relation. The virtual relation is the union of all these fragments.

The latter observation has an interesting consequence: a device database is internally represented as a distributed database. Virtual relations are partitioned across a set of devices. Base relations are stored on the database front-end. Distributed query

[2] We assume without loss of generality that a device function has exactly one return value; an extension to the general case is straightforward.

processing techniques are not implemented in the initial version of COUGAR; their design and integration is the main goal of COUGAR V2 that we are currently implementing.

Query Execution Plan

Virtual relations appear in the query execution plan at the same level as base relations. Base relations are accessed through (indexed) scans. Each virtual relation fragment is accessed on sensors using a virtual scan. A virtual scan incorporates in the query execution plan the propagation mechanism necessary to support long-running queries.

Our notion of virtual scan over a virtual relation fragment is similar to the fetch_wait cursor over an active relation in the Alert database system [18]. A fetch_wait cursor provides a blocking read behavior. This fetch_wait cursor returns new records as they are inserted in the active relation and blocks when all records have been returned. A classical cursor would just terminate when all records currently in the relation have been returned.

The join between a base relation and a virtual relation is basically a nested loop with a pipelined access to the virtual scans that encapsulate the execution of the sensor ADT function. (Note that we make the simplifying assumption that arguments to the sensor ADT functions are constants.) Indeed, the sensor ADT function is applied with identical parameters on all sensors involved in the query. The algorithm is presented below.

```
In: Base relation R, sensor ADT function f

Out: join between relation R and virtual relation associated to f

Initialize virtual scans for the virtual relation fragments
associated to f on all sensors involved in the query

FOREVER DO

   Get next output from the sensor virtual scan

   Find a matching sensor id in the base relation R

   If match is found then return record

ENDLOOP
```

The incremental results produced by a virtual join are directly transmitted to the client, or they are pipelined to the root of the execution plan (as the outer child in a nested loop join for instance[3]). Consequently, queries with relational aggregates or 'ORDER BY' clauses do not return an incremental result. Indeed, such queries require an operator to accumulate all the results produced by its children. With such operators no incremental results are produced before the query is stopped.

[3] Note that queries with sensor ADT functions applied on more than one collection of sensors require that the join between two virtual joins is a double-pipelined join.

3.3 Conclusions

Here are the conclusions that we have drawn from our experience with the initial version of COUGAR:

1. Representing stored data as relations with an ADT attribute representing sensors and sensor data as the output of ADT functions is a natural way of representing a sensor database.
2. Virtual joins are an effective way of executing ADT functions that do not return a value in a timely fashion (because they are often asynchronous, because they generally incur high latency or because they return multiple values over time).
3. Representing all signal processing functions as scalar functions fails to capture the ordering of sensor data in time. As a result, queries involving aggregates over time windows or correlations are difficult to express. This problem has previously been identified in the context of financial data [21].

4 Related Work

Two projects are representative of the efforts that are made to build wireless sensor network infrastructures: The WINS project at UCLA [17] and the Smart Dust project at UC Berkeley [14]. The model of sensor database that we introduce in Section 2 is applicable to both types of sensor networks. The COUGAR system is implemented on top of the WINS infrastructure.

The goals of the DataSpace project at Rutgers University are quite similar to the goals of a sensor database system [9]. Imielinski et al. recognized the advantages of the distributed approach over the warehousing approach for querying physical devices. In a DataSpace, devices that encapsulate data can be queried, monitored and controlled. Network primitives are developed to guarantee that only relevant devices are contacted when a query is evaluated. We are currently integrating COUGAR with similar networking primitives, i.e., the Declarative Routing Protocol developed at MIT-LL [5], and the SCADDS diffusion-based routing developed at ISI [10]. Other related projects include the TELEGRAPH project at UC Berkeley [1], which studies adaptive query processing techniques, and the SAGRES project at the University of Washington [11], which explores the use of data integration techniques in the context of device networks.

The environment of a sensor network with computing power at each node resembles a mobile computing environment [8]. Sensors differ from mobile hosts in that sensors only serve external requests but do not initiate requests themselves. Also, recent work on indexing moving objects, e.g. [16], is relevant in such environments. The techniques proposed however assume a centralized warehousing approach.

Our definition of sensor queries bears similarities with the definition of continuous queries [23]. Continuous queries are defined over append-only relations with time-stamps. For each continuous query, an incremental query is defined to retrieve all answers obtained in an interval of t seconds. The incremental query is issued repeatedly, every t seconds, and the union of the answers it provides constitute the answer to the continuous query. Instead of being used to maintain a persistent view,

incremental results are directly returned to users. The answers returned by the initial prototype of COUGAR respect the continuous queries semantics.

Time series can be manipulated in object-relational systems such as Oracle [16] or in array database systems such as KDB [13]. These systems do not support the execution of long-running queries over sequences.

5 Conclusion

We believe that sensor database systems are a promising new field for database research. We described a data model and long-running query semantics for sensor database systems where stored data are represented as relations and sensor data are represented as sequences. The version of the Cornell COUGAR system that we presented is a first effort towards such a sensor database system. The second version of COUGAR [4] improves on the initial prototype in that sequences are explicitly represented. This allows for more expressive sensor queries. In particular, queries containing aggregates over time windows can be expressed.

This first generation of the Cornell COUGAR systems demonstrated that the application of database technology shows much promise for providing flexible and scalable access to large collections of sensors. It also helped us identify a set of challenging issues that we are addressing with our ongoing research:

- Due to the large scale of a sensor network, it is highly probable that some of the sensors and some of the communication links will fail at some point during the processing of a long-running query. We are studying how sensor database systems can adjust to communication failures and return a more accurate answer at the cost of increased response time and resource usage.
- Sensor Data are measurements not facts. Indeed, to each value produced by a sensor is associated a probability that this value is correct. Often, sensor values correspond to continuous distributions, e.g. a normally distributed probability with a given mean and standard deviation. We are defining a data model and analogs of the relational operators for representing and manipulating continuous distributions.
- Because of the large scale and dynamic nature of a sensor network, we cannot assume that a centralized optimizer maintains global knowledge and thus precise meta-information about the whole network. We are studying how to maintain meta-data in a decentralized way and how to utilize this information to devise good query plans.

Acknowledgements
We would like to thank Tobias Mayr and Raoul Bhoedjiang who helped debug earlier versions of this paper. This paper benefited from interactions with the SensIT community. In particular, Bill Kaiser provided valuable information concerning the Sensoria WINS network. Tok Wee Hyong has implemented most of the sequence ADT extension for COUGAR V2. Joe Hellerstein suggested the relevance of sequences for sensor databases. This work is sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Material Command, USAF, under agreement number F-30602-99-0528.

References

1. Ron Avnur, Joseph M. Hellerstein: Eddies: Continuously Adaptive Query Processing. SIGMOD Conference 2000: 261-272
2. Ph. Bonnet, P.Seshadri. Device Database Systems. Proceedings of the International Conference on Data Engineering ICDE'99, San Diego, CA, March, 2000.
3. Ph.Bonnet, J.Gehrke, P.Seshadri. Querying the Physical World. IEEE personal Communications. Special Issue "Networking the Physical World". October 2000.
4. Ph.Bonnet, J.Gehrke, P.Seshadri. Towards Sensor Database Systems. Cornell CS Technical Report TR2000-1819. October 2000
5. D.Coffin, D.Van Hook, S.McGarry, S.Kolek. Declarative AdHoc Sensor. SPIE Integrated Command Environments. 2000.
6. D.Estrin, R.Govindan, J.Heidemann (Editors): Embedding the Internet. CACM 43(5) (2000)
7. The Intel Computing Continuum Conference, San Francisco, May, 2000. http://www.intel.com/intel/cccon/
8. Tomasz Imielinski, B. R. Badrinath: Data Management for Mobile Computing. SIGMOD Record 22(1): 34-39 (1993)
9. Tomasz Imielinski, Samir Goel: DataSpace - Querying and Monitoring Deeply Networked Collections in Physical Space. MobiDE 1999: 44-51
10. C.Intanagonwiwat, R.Govindan, D.Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. Mobicom'00.
11. Z. G. Ives, A. Y. Levy, J. Madhavan, R. Pottinger, S. Saroiu, I. Tatarinov, S. Betzler, Q. Chen, E. Jaslikowska, J. Su, W. Tak and T.Yeung: Self-Organizing Data Sharing Communities with SAGRES. SIGMOD Conference 2000: 582
12. Kx Systems Home Page: http://www.kx.com.
13. H. V. Jagadish, Inderpal Singh Mumick, Abraham Silberschatz: View Maintenance Issues for the Chronicle Data Model. PODS 1995: 113-124
14. J. M. Kahn, R. H. Katz and K. S. J. Pister, "Mobile Networking for Smart Dust", ACM/IEEE Intl. Conf. on Mobile Computing and Networking (MobiCom 99), Seattle, WA, August 17-19, 1999
15. Oracle8™ Time Series Data Cartridge. 1998. http://www.oracle.com/
16. Dieter Pfoser, Christian S. Jensen, Yannis Theodoridis: Novel Approaches in Query Processing for Moving Objects. VLDB 2000:
17. G.Pottie, W. Kaiser: Wireless Integrated Network Sensors (WINS): Principles and Approach. CACM 43(5) (2000)
18. U. Schreier, H. Pirahesh, R. Agrawal, C. Mohan: Alert: An Architecture for Transforming a Passive DBMS into an Active DBMS. VLDB 1991: 469-478
19. Praveen Seshadri, Miron Livny, Raghu Ramakrishnan: SEQ: A Model for Sequence Databases. ICDE 1995: 232-239
20. P. Seshadri. Enhanced Abstract Data Types in Object-Relational Databases. VLDB Journal 7(3): 130-140 (1998).
21. D.Shasha: Time Series in Finance: The Array Database Approach. 1998. http://cs.nyu.edu/shasha/papers/jagtalk.html
22. D.Tennenhouse: Proactive Computing. CACM 43(5) (2000)
23. Douglas B. Terry, David Goldberg, David Nichols, Brian M. Oki: Continuous Queries over Append-Only Databases. SIGMOD Conference 1992: 321-330

Appendix C:

GADT: A Probability Space ADT for Representing and Querying the Physical World*

Anton Faradjian, Johannes Gehrke
Department of Computer Science
Cornell University
{tony,johannes}@cs.cornell.edu

Philippe Bonnet†
Datalogisk Institut
Københavns Universitet
bonnet@diku.dk

Abstract

Large sensor networks are being widely deployed for measurement, detection, and monitoring applications. Many of these applications involve database systems to store and process data from the physical world. This data has inherent measurement uncertainties that are properly represented by continuous probability distribution functions (pdf's). We introduce a new object-relational data type, the Gaussian ADT GADT*, that models physical data as gaussian pdf's, and we show that existing index structures can be used as fast access methods for* GADT *data. We also present a measure-theoretic model of probabilistic data and evaluate* GADT *in its light.*

1 Introduction

Networks of radar, sonar, seismic, and thermal sensors are being deployed widely for measurement, detection, and monitoring applications. These sensor networks will create a flood of observational data of unprecedented scale [EGHK99]. Similarly, enormous quantities of physical data are, and will continue to be, generated by astronomical sky surveys [SKT+00]. A large class of these applications rely on database systems to store, filter, compare and aggregate large volumes of physical data [BS00].

Inherent to data that result from a physical measurement is *uncertainty* regarding the true value of the measured quantity. This uncertainty can properly be described by a *continuous probability distribution function (*$\mathrm{p.d.f.}$*)* over the possible measurement values. For example, consider a temperature sensor in your office that reports an estimate $\widehat{T}$ of the current temperature T; let this estimate be $\widehat{T} = 68°$ Fahrenheit (F). Given this measurement, do we believe that the temperature in your office is exactly $68°$ F? Assuming that the error introduced by the sensor has a gaussian distribution with a known standard deviation of $\sigma°$F, we can compute the probability that the true temperature T lies in the range $[T_1, T_2]$. In the context of a database application, a user should be able to submit a query that retrieves all temperatures whose true values lie in the range $[T1, T2]$ with a given probability p.

Note that we need to manage such uncertainties using probability theory, and not using fuzzy theory. There is no question here about fuzzy set membership or the definition of vague terms such as "tall" or "hot." Since the nature of our problem is fundamentally probabilistic, fuzzy relational models do not apply in our setting. [AR84, KF88, RM88].

In order to manage the uncertainty associated with physical data but at the same time take advantage of features of a modern database system, we need a data model for representing continuous $\mathrm{p.d.f.}$'s such as gaussians. Surprisingly, none of the numerous probabilistic data models described in the literature handles continuous $\mathrm{p.d.f.}$'s—all models deal with discrete $\mathrm{p.d.f.}$'s [CP87, BGMP92, DS96].

In this paper, we develop a data model for continuous $\mathrm{p.d.f.}$'s. Our first contribution is GADT, a concrete abstract datatype (ADT) for representing one-dimensional gaussian distributions. GADT is simple and expressive. We show that GADT is easy to implement as an extension to an existing object-relational DBMS, and we outline how we can access GADT data efficiently using indexing by linear constraint (QBLC) [GRSY97]. As a proof of concept, we have carried out a prototype implementation of GADT in the Cornell Predator ORDBMS [Ses98].

Our second contribution is a study of the theoretical aspects of probability space ADT's. Having started with the datatype GADT, we lift our level of abstraction to a measure-theoretic framework to reason about properties of datatypes that represent continuous as well as discrete probability distribution functions. We introduce probability spaces and events as the basic elements of any probabilistic data type.

*This work was supported by DARPA under contract F30602-99-2-0528, an IBM Faculty Development Award, and by gifts from Microsoft and Intel.

†Work done while at Cornell University.

We show that equality raises an interesting challenge for continuous distributions, and we introduce operations that overcome this challenge. This conceptual study does not only provide a framework for the future development of probabilistic ADT's, but also sheds light on several aspects of our one-dimensional gaussian model. Thus our measure-theoretic framework is not only an abstraction of given instantiations of probabilistic ADTs, it allows us to gain insights into the general functionality and methods that instantiations of probabilistic ADTs should encompass and what their semantics should be. The reader should therefore understand this paper as a trail that starts with a concrete instantiation, climbs to the abstract level, and then returns to the concrete instantiation with some insights from the abstract level.

The milestones along our trail are as follows. Section 2 introduces the gaussian ADT GADT and its methods. Section 3 outlines techniques for query processing using GADT data and queries. Section 4 studies the theoretical aspects of probability space ADTs, and Section 5 discusses the insights that our theoretical framework provides with respect to GADT. We discuss related work in Section 6 and conclude in Section 7.

2 GADT: The Gaussian ADT

In this section, we introduce GADT, the gaussian ADT with which we can represent physical measurements as continuous gaussian $\mathrm{p.d.f.}$'s. We first introduce gaussian $\mathrm{p.d.f.}$'s formally in Section 2.1. Section 2.2 introduces GADT, and Section 2.3 introduces the methods that GADT supports.

2.1 Preliminaries

A gaussian $\mathrm{p.d.f.}$ has the form

$$g_{\mu,\sigma}(x) \stackrel{def}{=} \frac{e^{-(x-\mu)^2/2\sigma^2}}{\sqrt{2\pi}\sigma} \tag{1}$$

The parameters μ and σ are the *mean* and *standard deviation* of the $\mathrm{p.d.f.}$, respectively. The definite integral of $g_{\mu,\sigma}$ is denoted $G_{\mu,\sigma}$ and gives the probability that the true value of x lies in the interval of integration:

$$G_{\mu,\sigma}([a,b]) \stackrel{def}{=} P\{x \in [a,b]\} = \int_a^b g_{\mu,\sigma}(t)\, dt. \tag{2}$$

For $z \geq 0$, we use $\epsilon(z)$ to denote $G_{0,1}([-z,z])$. It is related to the well-known error function $Erf(z)$ [Fel66, Vol. 1, Ch. 7]:

$$\epsilon(z) \stackrel{def}{=} G_{0,1}([-z,z]) = Erf\left(\frac{z}{\sqrt{2}}\right), \quad z \geq 0. \tag{3}$$

The function ϵ has an inverse, ϕ, defined on $[0,1]$ by:

$$\phi(\epsilon(z)) \stackrel{def}{=} z. \tag{4}$$

Both ϵ and ϕ are monotonically increasing.

2.2 The GADT Model

A measurement that is subject to many small and random errors is normally distributed and characterized by a gaussian $\mathrm{p.d.f.}$ A finite number of repetitions of a measurement also results in a normal distribution [Tay82]. We desire a data model that treats gaussians as first-class data values. GADT accomplishes this by defining a *gaussian ADT*: an instance of the ADT corresponds to a gaussian $\mathrm{p.d.f.}$, and, in terms of physical data representation, consists simply of the two real numbers μ and σ. GADT instances are by definition probabilistically independent of each other so that the joint $\mathrm{p.d.f.}$ of two gaussian instances is simply the product of their $\mathrm{p.d.f.}$'s. Statistical dependence between measured quantities can be represented using higher-dimensional gaussians; higher-dimensional gaussians are a topic for future research.

In order to evaluate the probability that a true physical value lies in a given interval, we need an *interval ADT*. The interval ADT represents intervals on the real line; it is ancillary to GADT. Due to space constraints, and for ease of explanation, we do not define the interval ADT formally, and we focus our attention on the case of single intervals, as opposed to unions of disconnected intervals.

We now use a simple denotational semantics to define GADT methods. The semantics make use of the following basic value mappings, which are generalized in Section 4. Given a GADT instance i having mean μ and standard deviation σ, we define the *gaussian mapping* $\mathcal{I}[\![i]\!]$ by

$$\mathcal{I}[\![i]\!] \stackrel{def}{=} G_{\mu,\sigma}. \tag{5}$$

Similarly, given an instance v of the interval ADT representing the real interval $[a,b]$, we define the *interval mapping* $\mathcal{V}[\![v]\!]$ by

$$\mathcal{V}[\![v]\!] \stackrel{def}{=} [a,b]. \tag{6}$$

Finally, given a real instance x representing the real number X, we define the *real mapping* $\mathcal{R}[\![x]\!]$ by

$$\mathcal{R}[\![x]\!] \stackrel{def}{=} X. \tag{7}$$

We use these three instance mappings to define the GADT methods $Prob$, $Diff$, and $Conf$, and we show how these methods can be used to pose queries involving data with continuous $\mathrm{p.d.f.}$'s.

2.3 GADT Methods

2.3.1 Selections with *Prob*

Computing the probability that a value lies inside an interval is the most fundamental GADT operation. The *Prob* ADT method provides this feature: It takes as argument an interval instance v and returns the probability that the true value of the measurement represented by a GADT instance i lies in $\mathcal{V}[\![v]\!]$:

$$\mathcal{R}[\![i.Prob(v)]\!] \stackrel{def}{=} \mathcal{I}[\![i]\!](\mathcal{V}[\![v]\!]). \quad (8)$$

Prob is useful for obtaining the likelihood of events. As an example, let R_1 be a relation having the GADT-valued attribute *Temp*, which stores a temperature measurement obtained from a temperature sensor. Using *Prob*, we can pose queries such as: Retrieve from R_1 all tuples whose *Temp* is within 0.5 degrees of 68 degrees with at least 60% probability:

SELECT *
FROM R_1
WHERE $R_1.Temp.Prob([67.5, 68.5]) \geq 0.6$

Another example is as follows: Retrieve from R_1 all tuples whose *Temp* is at least 75 degrees with probability at most 90%:

SELECT *
FROM R_1
WHERE $R_1.Temp.Prob([75, \infty]) \leq 0.9$

2.3.2 Comparisons with *Diff*

Another important operation is to compute the difference between two gaussians [Tay82]. Let i_1, i_2 be two GADT instances representing uncertain scalar quantities x_1 and x_2, respectively, and let $\mathcal{I}[\![i_1]\!] = g_{\mu_1,\sigma_1}(x_1)$ and $\mathcal{I}[\![i_2]\!] = g_{\mu_2,\sigma_2}(x_2)$. Because i_1 and i_2 are probabilistically independent, the p.d.f. of $x_1 - x_2$ is a gaussian g_{μ_-,σ_-}, where

$$\mu_- = \mu_1 - \mu_2, \text{ and} \quad (9)$$

$$\sigma_- = \sqrt{\sigma_1^2 + \sigma_2^2}. \quad (10)$$

We use DIFF to denote the difference between two gaussians:

$$\text{DIFF}(G_{\mu_1,\sigma_1}, G_{\mu_2,\sigma_2}) \stackrel{def}{=} G_{\mu_-,\sigma_-}.$$

Note that DIFF is not symmetric in its arguments. The *Diff* method computes DIFF:

$$\mathcal{I}[\![i_1.Diff(i_2)]\!] \stackrel{def}{=} \text{DIFF}(\mathcal{I}[\![i_1]\!], \mathcal{I}[\![i_2]\!]). \quad (11)$$

When used with *Prob*, *Diff* allows us to compare i_1 and i_2 by computing the probability that $a < x_1 - x_2 < b$:

$$P\{a < x_1 - x_2 < b\} = (i_1.Diff(i_2)).Prob([a, b]). \quad (12)$$

As an example, let R_1 and R_2 be two relations each having the GADT-valued attribute *Temp*, which stores a temperature. Consider the following query: Join R_1 and R_2 on the condition that $R_1.Temp$ is within 0.1 degrees of $R_2.Temp$ with probability at least 75%:

SELECT *
FROM R_1, R_2
WHERE $(R_1.Temp.Diff(R_2.Temp)).$
$Prob([-0.1, 0.1]) \geq 0.75$

2.3.3 Comparisons with *Conf*

In the context of astronomical data, C. Page shows that it is useful to compare gaussians by testing whether their confidence intervals overlap [Pag96]. Page calls this kind of join a "fuzzy join"[1] and recommends that it be implemented in all astronomical DBMS's. GADT provides the method *Conf* to do this. Given a GADT instance i and a probability $p \in [0, 1]$, $i.Conf(p)$ evaluates to the $100p$ % confidence interval. Specifically, if $\mathcal{I}[\![i]\!] = g_{\mu,\sigma}$, then

$$\mathcal{V}[\![i.Conf(p)]\!] \stackrel{def}{=} [\ \mu - \sigma \cdot \phi(p),\ \mu + \sigma \cdot \phi(p)\ ] \quad (13)$$

(recall the definition of ϕ from Equation 4). Let S_1, S_2 be two relations each having the GADT-valued attribute *Pos*, which stores the positions of stars along a certain dimension. Then we can ask the following query: Join S_1, S_2 on the condition that the 30% confidence interval of $S_1.Pos$ intersects the 35% confidence interval of $S_2.Pos$:[2]

SELECT *
FROM S_1, S_2
WHERE $S_1.Pos.Conf(0.3) \cap$
$S_2.Pos.Conf(0.35) \neq \emptyset$

2.4 Implementation

As a proof of concept, we performed a prototype implemention of GADT as an extension to the Cornell Predator object-relational DBMS [SP97]. We defined new abstract data types (ADTs) for gaussians and for intervals. We implemented the *Prob* and *Conf* methods of GADT. The computation of probabilities in *Prob* relies on an approximation of ϵ [FGB01]. An alternative is to rely on a pre-packaged implementation of ϵ, such as those found in Mathematica, Matlab, or the GNU C Compiler. The interval ADT is used to express ranges and the results of calls to the *Conf* method. In order to implement the Page join (in Section 2.3.3) we implemented a simple *Intersect* method that computes the intersection of two intervals. The gaussian and interval ADTs extend Predator's type subsystem;

[1] Arguably a misnomer, since it involves no fuzzy set theory.

[2] The interval ADT is assumed to provide a method to compute intersections of intervals. We use $\cap$ as informal notation for that method.

they do not rely on any features particular to this system and thus can be implemented in any ORDBMS.

3 Indexing GADT Relations

When dealing with large volumes of GADT data, queries cannot be efficiently processed by naively scanning relations; we need efficient access methods. Fast access to GADT data can be achieved by translating GADT queries into *queries by linear constraints (QBLC)*. Goldstein et al. [GRSY97] and Agarwal et al. [AAE98] have recently shown that QBLC can be processed efficiently using standard indexing structures such as the R-tree.

Let i be a GADT instance with $\mathcal{I}[\![i]\!] = G_{\mu,\sigma}$. Then i is logically equivalent to the pair (μ, σ), and any condition imposed on i is equivalent to a constraint on (μ, σ). If the condition is given by a boolean predicate b then we can visualize all instances satisfying b as a region B in the $\mu - \sigma$ plane (a subset of $\mathsf{R} \times \mathsf{R}^+$). We call B the *valid region of* b. We call any superset of B a *safe region for* b. A simple procedure for GADT query processing is the following two-stage process:

1. Compute a safe region that is expressible as a set of linear constraints; then
2. Use the constraints as input to a QBLC indexing engine such as the R-tree variant of Goldstein et al. [GRSY97].

Examples of safe region computation follow in the remainder of this section. More general questions of query processing and optimization for GADT are beyond the scope of this paper and await future research.

3.1 Safe regions for $Prob$

Here we show how to process efficiently a selection on the predicate

$$R.a.Prob(I) \geq p, \tag{14}$$

where I is the interval $I = [L, R]$. The adaptation of the procedure to other kinds of predicates is straightforward. Recall from Section 2 that ϵ and ϕ are related to the well-known error function.

3.1.1 Semi-infinite intervals

Suppose first that I is a semi-infinite interval. Without loss of generality, say $L = 0$ and $R = \infty$. We distinguish two cases: $p \geq 0.5$ and $p < 0.5$.

Case 1: $p \geq 0.5$. By the symmetry of gaussians, we must have $\mu \geq 0$. Then

$$G_{\mu,\sigma}([0, \infty]) = \frac{1}{2}(1 + \epsilon(\mu/\sigma)),$$

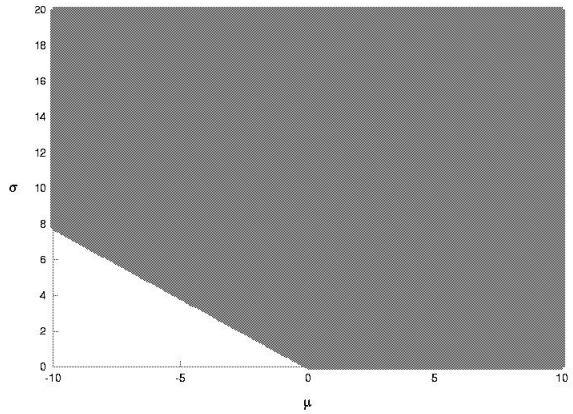

Figure 1. Valid region for $I = [0, \infty], p \geq 0.1$.

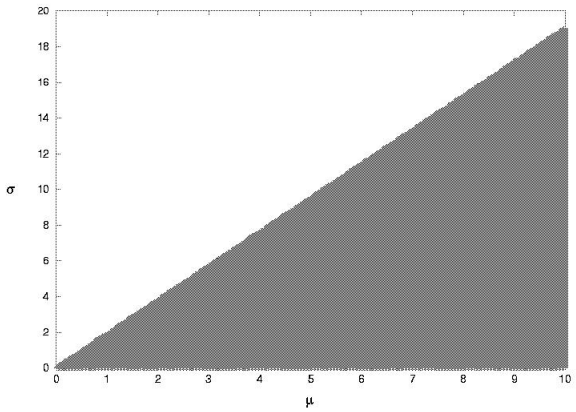

Figure 2. Valid region for $I = [0, \infty], p \geq 0.7$.

so Inequality 14 is equivalent to

$$2p \leq 1 + \epsilon(\mu/\sigma).$$

Since $\phi(x)$ is a monotonically increasing function of x, this last inequality is equivalent to

$$\mu \geq \sigma \cdot \phi(2p - 1). \tag{15}$$

Case 2: $p < 0.5$. Reasoning similarly to Case 1, we obtain

$$\mu \geq -\sigma \cdot \phi(1 - 2p). \tag{16}$$

Inequalities 15 and 16 are linear constraints that define exactly the valid region for the atomic predicate of Inequality 14 for the cases $p \geq 0.5$ and $p < 0.5$, respectively. Examples are shown in Figures 1 and 2.

3.1.2 Finite intervals

Suppose, without loss of generality, that $I = [-w, w]$ for some $w \geq 0$. We can again distinguish the two cases $p \geq 0.5$ and $p < 0.5$. It turns out that they too give rise to qualitatively different valid regions. But for finite intervals, the valid regions are not given by linear constraints. Consider, for example, the case $p \geq 0.5$. In order for $G_{\mu,\sigma}$ to lie in the valid region, μ must lie in the open interval $(-w, w)$. Inequality 14 is then equivalent to

$$\epsilon\left(\frac{|\mu| + w}{\sigma}\right) + \epsilon\left(\frac{|\mu| - w}{\sigma}\right) \geq 2p.$$

Because $\phi(x)$ is a nonlinear function of x, we cannot obtain a linear constraint involving μ and σ by applying ϕ to both sides of the last inequality, as we did above. Figures 3 and 4, which show plots of valid regions for $p = 0.1$ and $p = 0.7$, illustrate that the valid regions are indeed nonlinear. Observe, however, that for any p the valid region is enclosed by a *bounding box* given by the four linear constraints

$$\mu \geq -\mu^\star, \quad \mu \leq \mu^\star, \quad \sigma \geq 0, \quad \text{and} \quad \sigma \leq \sigma^\star. \tag{17}$$

The parameters $\mu^\star$ and $\sigma^\star$ are functions of p. We obtain $\sigma^\star$ by noting that, for fixed σ, $G_{\mu,\sigma}([-w, w])$ is maximum at $\mu = 0$. This follows from the symmetry of $g_{\mu,\sigma}$, and is illustrated in Figures 3 and 4. The parameter $\sigma^\star$ is therefore defined by the equation

$$G_{0,\sigma^\star}([-w, w]) \;=\; \epsilon(w/\sigma^\star) \;=\; p, \quad p \in [0, 1],$$

which is equivalent to

$$\sigma^\star = \frac{w}{\phi(p)}. \tag{18}$$

As for $\mu^\star$, we distinguish the two cases $p \geq 0.5$ and $p < 0.5$. When $p \geq 0.5$, $\mu^\star$ is easily seen to be w itself. When $p < 0.5$, however, the situation becomes more interesting. Consider, without loss of generality, a gaussian $G_{\mu,\sigma}$ with $\mu > w$. In the limit both of very large *and* of very small σ, we have $G_{\mu,\sigma}([-w, w]) = 0$. There is therefore a unique value of σ, which we denote σ_{max}, and which is a function of μ and w, that maximizes $G_{\mu,\sigma}([-w, w])$. Formally, $\sigma_{max}$is defined by

$$\left[\frac{\partial}{\partial\sigma} G_{\mu,\sigma}([-w, w])\right]_{\sigma=\sigma_{max}} \;=\; 0\,, \quad (\mu > w),$$

whose solution is

$$\sigma_{max} = \sqrt{\frac{2\mu w}{\ln\left(\frac{\mu+w}{\mu-w}\right)}}. \tag{19}$$

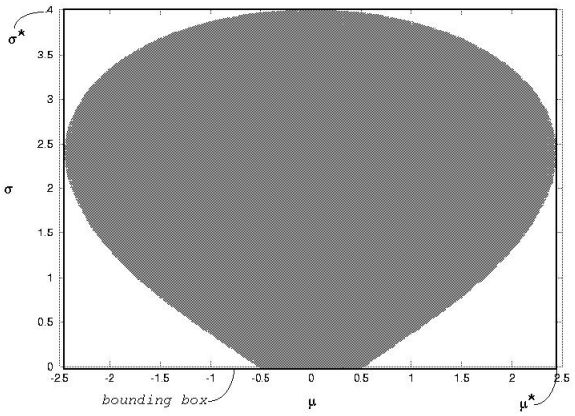

Figure 3. Valid region for $I = [-0.5, 0.5], p \geq 0.1$**.**

The maximum of $G_{\mu,\sigma}([-w, w])$, a function of μ and w, is denoted $G^w_{max}(\mu)$:

$$\begin{aligned} G^w_{max}(\mu) &\stackrel{def}{=} G_{\mu,\sigma_{max}}([-w, w]) \\ &= \frac{1}{2}\left[\epsilon\left(\frac{\mu + w}{\sigma_{max}}\right) - \epsilon\left(\frac{\mu - w}{\sigma_{max}}\right)\right] \\ &= \frac{1}{2}\left[\epsilon\left(\sqrt{\alpha\beta} + \sqrt{\frac{\alpha}{\beta}}\right)\right. \\ &\qquad \left. - \epsilon\left(\sqrt{\alpha\beta} - \sqrt{\frac{\alpha}{\beta}}\right)\right], \end{aligned}$$

where

$$\beta \stackrel{def}{=} \frac{\mu}{w}\,, \quad \alpha \stackrel{def}{=} \ln\left(\sqrt{\frac{\beta+1}{\beta-1}}\right).$$

There are infinitely many values of μ which are so large that no value of σ can satisfy Inequality 14. Such μ satisfy $G^w_{max}(\mu) < p$. But, in the upper halfplane U, there is clearly a unique μ that gives $G^w_{max}(\mu) = p$. This is $\mu^\star$. Formally,

$$G^w_{max}(\mu^\star) = p. \tag{20}$$

To the best of our knowledge, Equation 20 does not admit an analytical (closed-form) solution for $\mu^\star$. The problem can be recast as that of finding the root (zero) of the function $\zeta(\beta) \stackrel{def}{=} G^w_{max}(\mu^\star) - p$. The function ζ is very shallow for large β, however, which suggests that a root-finding algorithm (such as a variant of Newton's method) will struggle to find a good solution if p is small. In other words, the problem is ill-conditioned in that regime. A better solution is simply to tabulate a few values of $G^w_{max}(\mu)$ and to use

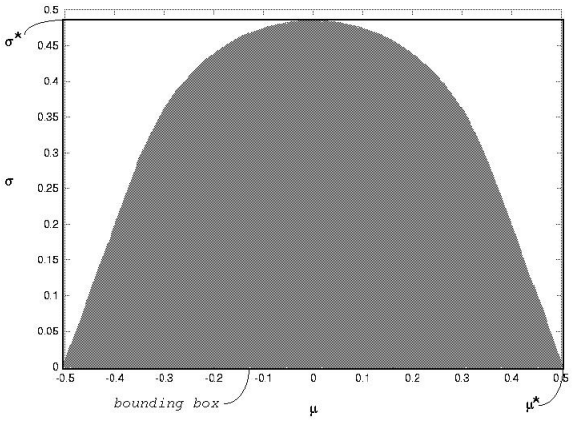

Figure 4. Valid region for $I = [-0.5, 0.5], p \geq 0.7$**.**

the resulting table to look up a conservative estimate of $\mu^\star$. The conservatism introduced decreases with increasing table size, but the table need not be very large; a few extra false positives would only negligibly impair performance. To improve on this scheme, we can interpolate the tabulated values using, for example, a multi-dimensional cubic spline.

3.2 Joins using *Diff*

The following query can be processed using index nested loops (INL) where the index is used to probe the inner relation S:

SELECT *
FROM R, S
WHERE $(R.a.Diff(S.b)).Prob([-w, w]) \geq p.$

Let t and s be tuples of R and S such that $t.a$ is the gaussian g_{μ_a,σ_a} and $s.b$ the gaussian g_{μ_b,σ_b}, respectively. Then $R.a.Diff(S.b)$ corresponds to the gaussian $g_d = g_{\mu_-,\sigma_-}$, from Equations 9 and 10. The results of Section 3.1.2 can thus be used with g_d as the gaussian in question. That is, Equations 17, 18, and 20, apply with the substitutions

$$\mu \to \mu_a - \mu_b, \quad \sigma \to \sqrt{\sigma_a^2 + \sigma_b^2}$$

which implies

$$\sigma_b \leq \sqrt{(\sigma^\star)^2 - \sigma_a^2}.$$

Since σ_a belongs to the outer tuple, it is a constant with respect to the inner index probe, and the last constraint is therefore linear in σ_b.

3.3 Safe regions for *Conf*

The test for confidence overlap also reduces to linear constraints on μ and σ. Let $g_1 = G_{\mu_1,\sigma_1}, g_2 = G_{\mu_2,\sigma_2}$ be two gaussians. Suppose we wish to test whether the c_1 confidence-interval C_1 of g_1 overlaps with the c_2 confidence-interval C_2 of g_2. For convenience, put $C_1 = [L_1, R_1]$ and $C_2 = [L_2, R_2]$. Then it is easy to show that:

$$\begin{aligned} L_1 &= \mu_1 - \phi(c_1)\sigma_1, \;\; R_1 = \mu_1 + \phi(c_1)\sigma_1, \\ L_2 &= \mu_2 - \phi(c_2)\sigma_2, \;\; R_2 = \mu_2 + \phi(c_2)\sigma_2. \end{aligned}$$

The condition for overlap is $L_1 \leq R_2 \wedge L_2 \leq R_1$, which is equivalent to

$$\mu_1 \leq R_2 + \phi(c_1)\sigma_1 \quad (21)$$
$$\mu_1 \geq L_2 - \phi(c_1)\sigma_1 \quad (22)$$

If we view g_2 as fixed, Equation 21 and Equation 22 are linear constraints involving μ_1 and σ_1, and the valid region is a curtailed cone.

4 Probability Space ADTs

Having presented an ADT for gaussian data, we now begin to explore a more general theory of probabilistic data in the ADT context. The goal is to define a framework (concepts, operations, ADT methods) that is independent of any particular $\mathrm{p.d.f.}$, so that we would not need to undertake a separate study for data whose uncertainty is given by other distributions, for example, a Gamma distribution. The model we present is not only $\mathrm{p.d.f.}$-agnostic, but also subsumes both continuous *and* discrete $\mathrm{p.d.f.}$'s under one general framework. To accomplish this goal, our model uses the language of measure theory.[3] In what follows, we use the term *probability space ADT (PSADT)* to refer to any datatype that aims to model probabilistic data using the ADT approach; the gaussian ADT GADT is an example of such a PSADT.

4.1 Spaces and Events

Let R be a database relation with an attribute a. Before attribute a can be typed as probabilistic and populated with PSADT instances, we must declare the *sample set* Ω in which may lie the true values of the quantities represented by those instances. This is analogous to specifying the domain of a regular attribute. Once Ω is specified, a PSADT instance m can then be modeled as a *probability measure* on the measurable space $(\Omega, \mathcal{F}_\Omega)$, where $\mathcal{F}_\Omega$ is a

[3] We assume the reader is already familiar with the basics of measure theory. See the books by Bartle [Bar95] or Billingsley [Bil95] for an introduction.

suitably chosen σ-algebra of subsets of Ω. We call a measurable set $E \in \mathcal{F}_\Omega$ an *event*, we call the measurable space $(\Omega, \mathcal{F}_\Omega)$ a *sample space*, and we call the triple $(\Omega, \mathcal{F}_\Omega, m)$ a *probability space*. The *domain* of attribute a is thus the set of all measures on the sample space $(\Omega, \mathcal{F}_\Omega)$. The *density* or *Radon-Nikodym derivative* of a probability measure with respect to some underlying measure on $(\Omega, \mathcal{F}_\Omega)$ is called a *probability density function* or *probability distribution function*, and is abbreviated "p.d.f."

As an example, if attribute a is to contain PSADT instances that represent uncertain integer data, then the domain of a is the set of all probability measures on the sample space $(\mathsf{Z}, 2^{\mathsf{Z}})$, where Z denotes the set of integers. In this example, the probability measures corresponding to PSADT instances will be *discrete*: the p.d.f.'s are with respect to the well-known counting measure [Bar95]. GADT, on the other hand, deals with *continuous* probability measures (see Section 5). Both examples fit neatly into the PSADT framework.

We assume that the DBMS supports the abstraction of a set, and we refer to it as the *event ADT*. It is ancillary to the PSADT, and should support basic set operations such as union, intersection, etc. In GADT the event ADT represented intervals over the real line. We need to generalize Equations 5 and 6 to accomodate the probability space abstraction. Given a PSADT instance i representing a measure m on a sample space $(\Omega, \mathcal{F}_\Omega)$, we define the *probability measure mapping* $\mathcal{I}[\![i]\!]$ by

$$\mathcal{I}[\![i]\!] \stackrel{def}{=} m. \tag{23}$$

Given an instance v of the event ADT representing an event $E \in \mathcal{F}_\Omega$, we define the *event mapping* $\mathcal{V}[\![v]\!]$ by

$$\mathcal{V}[\![v]\!] \stackrel{def}{=} E. \tag{24}$$

Unless stated otherwise, we assume throughout this section that the sample space is $(\Omega, \mathcal{F}_\Omega)$. We can now define PSADT methods.

4.2 Event probabilities

The most fundamental operation we can perform with probability spaces is to evaluate the probability assigned by a measure to an event. Let i be a PSADT instance with $\mathcal{I}[\![i]\!] = m$. Let v be an event instance with $\mathcal{V}[\![v]\!] = E$. Then the probability of E under m is given by

$$P\{E\} = m(E).$$

Accordingly, the most basic method of a PSADT is $Prob$, which takes an event instance as argument and computes its probability under a PSADT instance. Formally,

$$\mathcal{R}[\![i.Prob(v)]\!] \stackrel{def}{=} \mathcal{I}[\![i]\!](\mathcal{V}[\![v]\!]), \quad \mathcal{V}[\![v]\!] \in \mathcal{F}_\Omega. \tag{25}$$

4.3 Conditional measures

Let i be a PSADT instance with $\mathcal{I}[\![i]\!] = m$, and let v be an event instance with $\mathcal{V}[\![v]\!] = F$ such that $m(F) > 0$. The conditional probability measure m_F gives the conditional probability of an event E, given the event F:

$$m_F(E) \stackrel{def}{=} \frac{m(E \cap F)}{m(F)}, \ \forall E \in \mathcal{F}_\Omega. \tag{26}$$

The conditional measure can be used for updating: if F is new information then m_F represents the updated probability measure. The PSADT method $Cond$ computes conditional measures:

$$\mathcal{I}[\![i.Cond(v)]\!] \stackrel{def}{=} \mathcal{I}[\![i]\!]_{\mathcal{V}[\![v]\!]}. \tag{27}$$

4.4 Marginalization

Let the sample set Ω consist of the (ordered) cross product $\Omega_1 \times \cdots \times \Omega_n$. We can project out certain dimensions, obtaining a *marginal measure*. Let S be a subset of $\{1, \ldots, n\}$. Without loss of generality, let $S = \{1, \ldots, k\}$ with $k \leq n$. Let m be a measure. Then the *projection of* m *on* S is denoted π_m^S and defined as

$$\pi_m^S(E) \stackrel{def}{=} m(\Omega_1 \times \cdots \times \Omega_k \times E), \tag{28}$$

$$\forall E \subset \Omega_{k+1} \times \cdots \times \Omega_n \text{ s.t. } \Omega_1 \times \cdots \times \Omega_k \times E \in \mathcal{F}_\Omega.$$

The PSADT method $Proj$ computes projections: given a PSADT instance i, we have[4]

$$\mathcal{I}[\![i.Proj(S)]\!] \stackrel{def}{=} \pi^S_{\mathcal{I}[\![i]\!]}. \tag{29}$$

4.5 Comparisons

PSADT must provide a way to compare instances. Such comparisons would be at the heart of natural joins, for example. It turns out that, if we desire a model that treats discrete and continuous measures on the same footing, then the most basic and familiar kind of comparison, equality, needs to be reexamined.

4.5.1 Similarity: A Generalization of Equality

Suppose x_1, x_2 are two uncertain quantities that are known to lie in Ω. Let $S_2 = (\Omega^2, \sigma^2(\mathcal{F}_\Omega))$ be the product sample space, where $\sigma^2(\mathcal{F}_\Omega) \stackrel{def}{=} \{E_1 \times E_2 \mid E_1, E_2 \in \mathcal{F}_\Omega\}$, and let m be a joint probability measure on S_2 for x_1 and x_2. The following discussion applies in either of the following two situations:

[4] For simplicity, we do without a "set mapping" that maps an instance of a set to the set it represents. Hence S appears on both sides of Equation 29.

- There are PSADT instances i_1, i_2 such that m is the product measure $m = \mathcal{I}[\![i_1]\!] \times \mathcal{I}[\![i_2]\!]$.[5]
- There is a single instance i such that $m = \mathcal{I}[\![i]\!]$. (This allows for the possibility of non-factorizable product measures, i.e., attributes that are not probabilistically independent.)

We wish to compute the probability $P\{x_1 = x_2\}$ that x_1 and x_2 are equal. We might proceed as follows. Let $\mathcal{Q} \subseteq \Omega^2$ be the equality relation

$$\mathcal{Q} \stackrel{def}{=} \{(x_1, x_2) \in \Omega^2 \mid x_1 = x_2\}$$

(and assume that $\mathcal{Q} \in \sigma^2(\mathcal{F}_\Omega)$). Then the probability that the values of x_1 and x_2 are equal is simply

$$P\{x_1 \, \mathcal{Q} \, x_2\} \stackrel{def}{=} P\{(x_1, x_2) \in \mathcal{Q}\} = m(\mathcal{Q}). \quad (30)$$

The problem with this approach is that it works when m is discrete but not when it is continuous: in the latter case $m(\mathcal{Q})$ is generally zero. This is just a multi-dimensional analogue of the familiar fact that, given a continuous $\mathrm{p.d.f.}$ on R, there is zero probability that the true value equals any one real number.

The natural solution is to generalize the notion of equality, by replacing $\mathcal{Q}$ with a larger relation $\mathcal{E} \subseteq \Omega^2$. $\mathcal{E}$ is the set of all pairs (x_1, x_2) such that x_1 and x_2 are considered to be *similar* to one another, in whatever sense is appropriate to the application at hand. We require that $\mathcal{E}$ be reflexive, symmetric, measurable (i.e., $\mathcal{E} \in \sigma^2(\mathcal{F}_\Omega)$), and a superset of $\mathcal{Q}$. We do not require that $\mathcal{E}$ be transitive. We call such a relation $\mathcal{E}$ a *similarity event*. It is a relation on Ω and an event in Ω^2. The probability of equality in Equation 30 then becomes a *probability of similarity under* $\mathcal{E}$:

$$P\{x_1 \mathcal{E} x_2\} = m(\mathcal{E}). \quad (31)$$

As an example of a similarity relation, suppose Ω is a metric space with metric d.[6] Let $x \in \Omega$. Then the *neighborhood of radius* r *centered on* x is denoted by $B_r(x)$ and defined by $B_r(x) \stackrel{def}{=} \{y \in \Omega \mid d(x, y) \leq r\}$. Let $\Delta : \Omega \to \mathsf{R}$ be a function, called a *radius function*. Let

$$\mathcal{E}_\Delta \stackrel{def}{=} \{(x, y) \in \Omega^2 \mid y \in B_{\Delta(x)}(x) \ \vee \ x \in B_{\Delta(y)}(y)\} \quad (32)$$

be a relation on Ω. For reasonable choices of the σ-algebra $\mathcal{F}_\Omega$, $\mathcal{E}_\Delta$ will be a measurable set and thus will be a similarity event. We call it *metric similarity under the radius function* Δ. The simplest radius function is constant:

$$\exists \, \delta \in \mathsf{R}, \ \forall x \in \Omega, \ \Delta(x) = \delta. \quad (33)$$

We can obtain a more sophisticated radius function if the metric space Ω is also a *norm space*, that is, a *vector space* with an associated *norm* $\| \cdot \| : \Omega \to \mathsf{R}$. The norm induces a metric $d_{\|\cdot\|}(x, y) \stackrel{def}{=} \|x - y\|, \ \forall x, y \in \Omega$. Now let $\alpha \in \mathsf{R}$ be a small positive real number. Then the radius function $\Delta_\alpha(x) \stackrel{def}{=} \alpha\|x\|$ gives rise to a metric similarity $\mathcal{E}_{\Delta_\alpha}$ (substitute Δ_α for Δ in Equation 32). This similarity event judges two quantities (vectors) to be similar if their difference is small compared to their norms.

Choosing a similarity relation involves a degree of arbitrariness and subjectivity. This is not surprising. A certain amount of subjectivity *should* be involved in deciding whether two real-valued attribute values are equal, even when there is no uncertainty associated with them. Is the difference between 1 and $1 + 10^{-6}$ so significant as to make the numbers unequal? Only the user can decide the answer, and the verdict will depend on the situation at hand. Users therefore rely on similarity relations even when the data are certain. The only additional restriction imposed in the presence of uncertainty is that the similarity relations be measurable.

4.5.2 Total Variation

Probability measures can be compared using the *total variation distance* (TVD). The TVD between the two measures m_1, m_2 is defined as half the measure assigned to the entire sample set Ω by the total variation of their difference [Bar95]:

$$\mathrm{TVD}(m_1, m_2) = \frac{1}{2}|m_1 - m_2|(\Omega).$$

TVD is symmetric in its arguments. It is used to quantify the difference between m_1 and m_2 *as measures*. An example of such a use can be found in Barbará et al. [BGMP92]. In some cases it is natural to interpret a small TVD as indicating a high probability of equality of the underlying data values. This is especially true in the case of physical measurement, where the $\mathrm{p.d.f.}$'s are gaussian and where their expected values are interpreted as best estimates of measurements. The TVD is also appealing because it is a metric and thus renders the vector space of measures a metric space.

The PSADT method TVD computes total variations:

$$\mathcal{R}[\![i.TVD(j)]\!] \stackrel{def}{=} \mathrm{TVD}(\mathcal{I}[\![i]\!], \mathcal{I}[\![j]\!]).$$

4.5.3 Confidence overlap

Probability measures often have canonical choices of *confidence sets*. For example, the interval $[-1, 1]$ is the 68% confidence interval of the gaussian $G_{0,1}$. We use CONF to denote this mapping, as in $\mathrm{CONF}(G_{\mu,\sigma}, 0.68) = [\mu - \sigma, \mu + \sigma]$. The notion of confidence gives rise to the following comparison. Let i_1, i_2 be PSADT instances with $\mathcal{I}[\![i_1]\!] = m_1$

[5] The $\times$ notation refers to the product measure; as always, we assume i_1 and i_2 are probabilistically independent.

[6] Recall that a set Ω is a metric space if there is a function $d : \Omega \to \mathsf{R}$, called a *metric on* Ω, satisfying: $d(x, x) = 0 \ \forall x \in \Omega$; $d(x, y) = d(y, x) \ \forall x, y \in \Omega$; and $d(x, z) \leq d(x, y) + d(y, z) \ \forall x, y, z \in \Omega$.

and $\mathcal{I}[\![i_2]\!] = m_2$. Given $p_1, p_2 \in [0,1]$, we say m_1 and m_2 are *confidence-equal* if their respective confidence sets intersect:

$$\text{CONF}(m_1, p_1) \cap \text{CONF}(m_2, p_2) \neq \emptyset.$$

Informally, for smaller p_1 and p_2, confidence-equality implies that the true values are "closer" to one another. As mentioned above, this is the comparison that underlies the "fuzzy join" proposed by Page [Pag96]. It is useful, for example, when joining astronomical tables based on the positions of stars. We emphasize that CONF is not defined for arbitrary probability measures, but usually only for measures with parametric p.d.f.'s.

The PSADT method *Conf* computes confidence sets:

$$\mathcal{V}[\![i.Conf(p)]\!] \stackrel{def}{=} \text{CONF}(\mathcal{I}[\![i]\!], \mathcal{R}[\![p]\!]). \tag{34}$$

5 GADT revisited

We now demonstrate that GADT is an instance of the foregoing model: it is a PSADT that represents gaussian p.d.f.'s with respect to the Lebesgue measure λ on the real line. Specifically, if $\mathcal{B}$ is the Borel σ-algebra on R, then the domain of a GADT attribute is the set of all measures on the sample space $(\mathsf{R}, \mathcal{B})$ conforming to the following two restrictions:

1. Only intervallic events are supported. Compare Equation 6 with Equation 24.
2. All p.d.f.'s are with respect to λ and are given by Equation 1.

As Section 2 shows, GADT is useful in spite of restriction 1. That is, we can pose many interesting queries without needing to take unions and intersections of intervals. However, restriction 2 gives rise to at least two complications. The first is the following complication with *Cond*. Given an instance i such that $\mathcal{I}[\![i]\!] = G_{\mu,\sigma}$, and given a conditioning interval v, the instance $j \equiv i.Cond(v)$ would store v in its state, along with μ and σ, and would implement *Prob* using Equation 26. The problem is that $\mathcal{I}[\![j]\!]$ is no longer gaussian, and, for example, the indexing techniques given in Section 3 would no longer apply. One solution is to use j only as an intermediate result and to forbid its being stored in a relation. This is clearly a disadvantage if we wish to use *Cond* to update the database. Restriction 2 also gives rise to a second complication that we deal with in Section 5.1.

GADT provides two ways to compare values: *Conf* and *Diff*. *Conf* is obviously the same ADT method as that discussed in Section 4.5.3 (compare Equation 13 with Equation 34). *Diff* can be understood as a change of variables followed by an integration [Tay82]. Referring to Equation 12, if $m = \mathcal{I}[\![(i_1.Diff(i_2))]\!]$, then $m([-\delta, \delta])$ corresponds to the probability assigned by the joint p.d.f. J to the set $\mathcal{E}_\delta = \{(x,y) \in \mathsf{R}^2 : |x-y| \leq \delta\}$. In other words, $\mathcal{E}_\delta$ is a similarity event, and *Diff* is used to implement metric similarity (Equation 33). For example, the query in Section 2.3.2 uses metric similarity with radius $\delta = 0.1$.

Thus, GADT implements the PSADT notions of event, σ-algebra, probability, comparison, and similarity, and it endows each of these with semantics specific to gaussians. It is an instance of a PSADT.

5.1 On the possibility of arithmetic operations

Having defined *Diff*, a method which computes the difference between two uncertain quantities, the reader may well ask if it is possible to define a method that computes the sum, or the product, or the quotient, or even an arbitrary scalar function of uncertain quantities. This would amount to an arithemtic of uncertain quantities and would provide a way to propagate uncertainties from simple PSADT instances to compound PSADT instances. It would also enable us to manipulate uncertain data as naturally as if they were simple numbers.

There is some hope of achieving such an arithmetic in GADT. To illustrate, let $f : \mathsf{R}^n \to \mathsf{R}$ be an n-ary scalar function, and let $x_1, \ldots, x_n$ be uncertain quantities with gaussian p.d.f.'s $g_{\mu_1,\sigma_1}, \ldots, g_{\mu_n,\sigma_n}$, respectively. It is a fact that, *if σ_k is small compared to μ_k for all $k \in \{1, \ldots, n\}$*, then $f(x_1, \ldots, x_n)$ is normally distributed with p.d.f. g_{μ_f,σ_f} [Tay82], where[7]

$$\mu_f = f(\mu_1, \ldots, \mu_n), \quad \text{and} \tag{35}$$

$$\sigma_f = \sqrt{\sum_{k=1}^{n} \left(\sigma_k \left(\frac{\partial f}{\partial x_k} \right)_{(x_1,\ldots,x_n)=(\mu_1,\ldots,\mu_n)} \right)^2}.$$

When $f(x_1, x_2) = x_1 \pm x_2$, Equation 35 is exact regardless of the size of μ_1/σ_1 and μ_2/σ_2 [Tay82]. But for general f the best we can hope for is accuracy to first-order in μ_k/σ_k. Such an arithmetic extension to GADT is therefore useful only in applications that do not require the exact computation of probabilities. This also suggests that the correct p.d.f. for f is not gaussian, so that GADT boundaries have already been crossed (restriction 2 in Section 5). Such complications are manageable, however, and we see that arithmetic for gaussians is feasible, but only because gaussians have many special properties. Thus, although we could try to generalize our attempts at arithmetic by defining arithmetic methods for non-gaussian p.d.f.'s, there is probably no way to implement them, because neat formulas such as Equation 35 probably do not exist for those p.d.f.'s. In conclusion, there is hope that arithmetic can work with GADT, but only because it is special. Whether arithmetic can also work with a general PSADT is a matter for future research.

[7] These formulae are used in scientific data analysis to propagate measurement error.

6 Related Work

Probabilistic data models (PDM's) have been investigated extensively in the literature, but to the best of our knowledge all of the previous models support only discrete p.d.f.'s. The beginnings of the field can be viewed as extensions of early work on data incompleteness [Lip79, IL84, AKG99].

Wong treats data values as random variables, and regards query processing on uncertain data as a matter of statistical inference [Won82]; his paper has strong connections to the ideas of Lipski [Lip79]. The PDM's of Cavallo and Pittarelli extend each record by a probability stamp such that the sum of all probability stamps over a relation equals one; thus a relation directly encodes a probability distribution [CP87, Pit94].

Lakshmanan and Sadri include probabilities into the rule system of a deductive database through an algebra of confidence-intervals and a probabilistic calculus [LS94]. They also provide results on soundness, completeness, termination, and complexity of their model. Lakshmanan et al. give a probablistic relational model that aims for maximum flexibility by supporting, multiple strategies for combining basic events into complex events [LLRS97].

There is also a considerable body of work on fuzzy relations [AR84, KF88, RM88]. A number of authors have already observed, however, that the fuzzy approach to uncertainty in data is signficantly different from the probabilistic approach [BGMP92, DS96, LLRS97]. Generally, fuzzy logic is not concerned with uncertainty, but with compensating for the lack of expressivity in a language.

The work of Barbará et al. has been particularly influential for us [BGMP92]. Their model represents a discrete probability distribution as a first-class value, in the form of a nested relation. Dey and Sarkar present a model that is a hybrid of the PDM's by Barbará et al. [BGMP92] and Cavallo and Pittarelli[CP87] (see [DS96]). Their relations incorporate probability stamps that are not required to add up to unity.

7 Conclusions

We introduced GADT, a new probabilistic ADT that is especially suitable for representing data in the emerging class of applications that monitor the physical world. Our solution relies on ORDBMS ADT technology and supports continuous p.d.f.'s. We also demonstrated that fast access methods exist for GADT. We believe that GADT is an important step towards general database support for data whose uncertainty is represented by continuous p.d.f.'s. We also presented the general notion of a probability space ADT (PSADT) and showed how GADT conforms to it. The PSADT model is defined in terms of measure-theory and thus encompasses both discrete and continuous p.d.f.'s.

This paper represents our intial work on probabilistic data models, and there are numerous avenues for future work:

- Physical measurements often involve more than one dimension. For instance, most astronomical data are represented as two-dimensional gaussians. We intend to study such multi-dimensional p.d.f.'s.
- Gaussians are not the only relevant p.d.f. for modelling physical measurements. For instance, heavy-tailed non-gaussian distributions have been introduced to model phenomena with impulsive background noise [Mid99]. For these reasons and those given in Section 5.1, we are interested in the challenge of supporting arbitrary p.d.f.'s.
- Since we are interested in sensor data reduction, we would like to extend the model by introducing aggregate operators.
- We are currently investigating how we can use GADT to represent the results of approximate query answers, where the uncertainty associated with query incompleteness combines with the uncertainty inherent in the measurement data.
- Interesting questions regarding the processing and optimization of general queries on uncertain data await further exploration.
- Since most continuous p.d.f.'s represent real-valued data, it is worth inquiring into the possibility of a general "probabilistic arithmetic."

Acknowledgments. We thank Alin Dobra, Alexandre Evfimievski, Adam Florence, Steve Vavasis, Divesh Srivastava, and Dexter Kozen for helpful discussions.

References

[AAE98] Pankaj K. Agarwal, Lars Arge, and Jeff Erickson. Efficient searching with linear constraints. In *PODS*, pages 169–178, 1998.

[AKG99] Serge Abiteboul, Paris C. Kanellakis, and Gösta Grahne. On the representation and querying of sets of possible worlds. *Theoretical Computer Science*, 78(1):158–187, 1999.

[AR84] M. Anvari and G. F. Rose. Fuzzy relational databases. In J. C. Bezdek, editor, *Proceedings of the 1st International Conference on Fuzzy Information Processing*, pages B–6–3. CRC Press, 1984.

[Bar95] Robert G. Bartle. *The Elements of Integration and Lebesgue Measure*. Wiley, 1995.

[BGMP92] Daniel Barbará, Hector Garcia-Molina, and Daryl Porter. The management of probabilistic data. *TKDE*, 4(5):487–502, 1992.

[Bil95] P. Billingsley. *Probability and Measure*. Wiley, 1995.

[BS00] Philippe Bonnet and Praveen Seshadri. Device database systems. In *ICDE 2000, San Diego, California, USA*, page 194. IEEE Computer Society, 2000.

[CP87] Roger Cavallo and Michael Pittarelli. The theory of probabilistic databases. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *VLDB 1987, Brighton, England*, pages 71–81. Morgan Kaufmann, 1987.

[DS96] Debabrata Dey and Sumit Sarkar. A probabilistic relational model and algebra. *TODS*, 21(3):339–369, 1996.

[EGHK99] Deborah Estrin, Ramesh Govindan, John Heidemann, and Satish Kumar. Next century challenges: scalable coordination in sensor networks. In *Proceedings of the fifth annual ACM/IEEE International Conference on Mobile Computing and Networking August 15 - 19, 1999, Seattle, WA USA*, pages 263–270, 1999.

[Fel66] W. Feller. *An Introduction to Probability Theory and its Applications*. Wiley, 1966.

[FGB01] Anton K. Faradjian, Johannes Gehrke, and Philippe Bonnet. A measure-theoretic probabilistic data model. Technical report, Cornell University, 2001.

[GRSY97] Jonathan Goldstein, Raghu Ramakrishnan, Uri Shaft, and Jie-Bing Yu. Processing queries by linear constraints. In *PODS*, pages 257–267, 1997.

[IL84] Tomasz Imieliński and Witold Lipski Jr. Incomplete information in relational databases. *Journal of the ACM*, 31(4):761–791, October 1984.

[KF88] G. J. Klir and T. A. Folger. *Fuzzy Sets, Uncertainty and Information*. Prentice Hall, New Jersey, 1988.

[Lip79] Witold Lipski Jr. On semantic issues connected with incomplete information databases. *TODS*, 4(3):262–296, 1979.

[LLRS97] Laks V. S. Lakshmanan, Nicola Leone, Robert Ross, and V. S. Subrahmanian. Probview: A flexible probabilistic database system. *TODS*, 22(3):419–469, 1997.

[LS94] Laks V. S. Lakshmanan and Fereidoon Sadri. Probabilistic deductive databases. In Maurice Bruynooghe, editor, *Logic Programming, Proceedings of the 1994 International Symposium, November 13-17, ISBN 0-262-52191-1*, pages 254–268, 1994.

[Mid99] D. Middleton. Non-gaussian noise models in signal processing for telecommunications: New methods and results for class a and class b noise models. *IEEE Transactions on Information Theory*, 45:1129–1149, May 1999.

[Pag96] Clive G. Page. Astronomical tables, 2-d indexing, and fuzzy-joins. In Per Svensson and James C. French, editors, *SSDBM 1996*, pages 44–52. IEEE Computer Society, 1996.

[Pit94] Michael Pittarelli. An algebra for probabilistic databases. *TKDE*, 6(2):293–303, 1994.

[RM88] K. V. S. V. N. Raju and Arun K. Majumdar. Fuzzy functional dependencies and lossless join decomposition of fuzzy relational database systems. *TODS*, 13(2):129–166, 1988.

[Ses98] Praveen Seshadri. Enhanced abstract data types in object-relational databases. *VLDB Journal*, 7(3):130–140, 1998.

[SKT+00] Alexandar Szalay, Peter Z. Kunszt, Ani Thakar, Jim Gray, and Donald R. Slutz. Designing and mining multi-terabyte astronomy archives: The sloan digital sky survey. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *SIGMOD 2000*, volume 29, pages 451–462. ACM, 2000.

[SP97] Praveen Seshadri and Mark Paskin. Predator: An ordbms with enhanced data types. In Joan Peckham, editor, *SIGMOD 1997*, pages 568–571. ACM Press, 1997.

[Tay82] John R. Taylor. *An Introduction to Error Analysis: The Study of Uncertainties in Physical Measurements*. University Science Books, 1982.

[Won82] Eugene Wong. A statistical approach to incomplete information in database systems. *TODS*, 7(3):470–488, 1982.

Appendix D:

Query Processing for Sensor Networks

Yong Yao Johannes Gehrke

Department of Computer Science
Cornell University
Ithaca, NY 14850
{yao,johannes}@cs.cornell.edu

Abstract

Hardware for sensor nodes that combine physical sensors, actuators, embedded processors, and communication components has advanced significantly over the last decade, and made the large-scale deployment of such sensors a reality. Applications range from monitoring applications such as inventory maintenance over health care to military applications.

In this paper, we evaluate the design of a *query layer* for sensor networks. The query layer accepts queries in a declarative language that are then optimized to generate efficient query execution plans with in-network processing which can significantly reduce resource requirements. We examine the main architectural components of such a query layer, concentrating on in-network aggregation, interaction of in-network aggregation with the wireless routing protocol, and distributed query processing. Initial simulation experiments with the ns-2 network simulator show the tradeoffs of our system.

1 Introduction

Recent developments in hardware have enabled the widespread deployment of sensor networks consisting of small sensor nodes with sensing, computation, and communication capabilities. Already today networked sensors measuring only a few cubic inches can be purchased commercially, and Moore's law tells us that we will soon see components that measure 1/4 of a cubic inch, running an embedded version of a standard operating system, such as an embedded version of Linux or Windows CE .NET [2, 1]. Figure 1 shows a Berkeley MICA Mote[13], one of the platforms available commercially today, and Figure 2 shows its hardware characteristics.[1] Sensor nodes come in a variety of hardware configurations, from nodes connected to the local LAN attached to permanent power sources to nodes communicating via wireless multi-hop RF radio powered by small batteries, the types of sensor nodes considered in this paper. Such sensor nodes have the following resource constraints:

- **Communication.** The wireless network connecting the sensor nodes provides usually only a very limited quality of service, has latency with high variance, limited bandwidth, and frequently drops packets. [28].
- **Power consumption.** Sensor nodes have limited supply of energy, and thus energy conservation needs to be of the main system design considerations of any sensor network application. For example, the MICA motes are powered by two AA batteries, that provide about 2000mAh [13], powering the mote for approximately one year in the idle state and for one week under full load.
- **Computation.** Sensor nodes have limited computing power and memory sizes. This restricts the types of data processing algorithms on a sensor node, and it restricts the sizes of intermediate results that can be stored on the sensor nodes.
- **Uncertainty in sensor readings.** Signals detected at physical sensors have inherent uncertainty, and they may contain noise from the environment. Sensor malfunction might generate inaccurate data, and unfortunate sensor placement (such as a temperature sensor directly next to the air conditioner) might bias individual readings.

Future applications of sensor networks are plentiful. In the intelligent building of the future, sensors are deployed in offices and hallways to measure temperature,

Proceedings of the 2003 CIDR Conference

[1] MICA motes are available from `www.xbow.com`.

Figure 1: A Berkeley MICA Mote

Processor	4Mhz, 8bit MCU (ATMEL)
Storage	512KB
Radio	916Mhz Radio (RF Monolithic)
Communication Range	100 ft
Data Rate	40 Kbits/sec
Transmit Current	12 mA
Receive Current	1.8 mA
Sleep Current	5 uA

Figure 2: Hardware Characteristics of a MICA Mote

noise, light, and interact with the building control system. People can pose queries that are answered by the sensor network, such as "Is Yong in his office", or "Is there an empty seat in the meeting room?" Another application is scientific research. As an example, consider a biologist who may want to know of the existence of a specific species of birds, and once such a bird is detected, the bird's trail should be mapped as accurately as possible. In this case, the sensor network is used for automatic object recognition and tracking. More specific applications in different fields will arise, and instead of deploying preprogrammed sensor networks only for specific applications, future networks will have sensor nodes with different physical sensors for a wide variety of application scenarios and different user groups.[2]

In this paper, we develop a query layer for wireless sensor networks. Our approach is motivated by the following three design goals. First, we believe that declarative queries are especially suitable for sensor network interaction: Clients issue queries without knowing how the results are generated, processed, and returned to the client. Sophisticated catalog management, query optimization, and query processing techniques will abstract the user from the physical details of contacting the relevant sensors, processing the sensor data, and sending the results to the user. Thus one of the main roles of the query layer is to process declarative queries.

Our second design goal is motivated by the importance of preserving limited resources, such as energy and bandwidth in battery-powered wireless sensor networks. Data transmission back to a central node for offline storage, querying, and data analysis is very expensive for sensor networks of non-trivial size since communication using the wireless medium consumes a lot of energy. Since sensor nodes have the ability to perform local computation, part of the computation can be moved from the clients and pushed into the sensor network, aggregating records, or eliminating irrelevant records. Compared to traditional centralized data extraction and analysis, In-network processing can reduce energy consumption and reduce bandwidth usage by replacing more expensive communication operations with relatively cheaper computation operations, extending the lifetime of the sensor network significantly. For example, the ratio of energy spent in sending one bit versus executing one instruction ranges from 220 to 2900 in different architectures [29].[3] Thus the second main role of the query layer is to perform in-network processing.

Different applications usually have different requirements, from accuracy, energy consumption to delay. For example, a sensor network deployed in a battlefield or rescue region may only have a short life time but a high degree of dynamics. On the other had, for a long-term scientific research project that monitors an environment, power-efficient execution of long-running queries might be the main concern. More expensive query processing techniques may shorten processing time and improve result accuracy, but might use a lot of power. The query layer can generate query plans with different tradeoffs for different users.

In this paper, we propose and evaluate a database layer for sensor networks; we call the component of the system that is located on each sensor node the *query proxy*. Architecturally, on the sensor node, the query proxy lies between the network layer and the application layer, and the query proxy provides higher-level services through queries.

Given the view of a sensor network as a huge distributed database system, we would like to adapt existing techniques from distributed and heterogeneous database systems for a sensor network environment. However, there are major differences between sensor networks and traditional distributed and heterogeneous database systems.

First, sensor networks have communication and computation constraints that are very different from regular desktop computers or dedicated equipment in data centers, and query processing has to be aware of these constraints. One way of thinking about such constraints is the analogous interaction with the file systems in traditional database systems [37]. Database systems bypass the file system buffer to have direct

[2]The MICA motes already support temperature sensors, light sensors, magnetometers, accelerometers, and microphones.

[3]This is only a rule of thumb, since transmission range, bit error rates, and instruction width influence this parameter significantly.

control over the disk. For a sensor network database system, the analogous counterpart is the networking layer, and for intelligent resource management we have to ensure that the query processing layer is tightly integrated with the networking layer. Second, the notion of the cost of a query plan has changed, as the critical resource in a sensor network is power, and query optimization and query processing have to be adapted to take this optimization criterion into account.

While developing techniques that address these issues, we must not forget that scalability of our techniques with the size of the network, the data volume, and the query workload is an intrinsic consideration to any design decision.

Overview of the paper. The remainder of the paper is structured as follows. In the next section, we introduce our model of a sensor network, sensor data, and the class of queries that we consider in this paper. We then demonstrate our algorithms to process simple aggregate queries with in-network aggregation (Section 3), and investigate the interaction between the routing layer and the query layer (Section 4). We discuss how to create query plans to evaluate more complicated queries, and discuss query optimization for specific types of queries (Section 5). In a thorough simulation study, we examine the performance of our approach, and compare and analyze the performance of different query plans (Section 6).

2 Preliminaries

2.1 Sensor Networks

A sensor network consists of a large number of sensor nodes [27]. Individual sensor nodes (or short, nodes) are connected to other nodes in their vicinity through a wireless network, and they use a multihop routing protocol to communicate with nodes that are spatially distant. Sensor nodes also have limited computation and storage capabilities: a node has a general-purpose CPU to perform computation and a small amount of storage space to save program code and data.

We will distinguish a special type of node called a *gateway node*. Gateway nodes are connected to components outside of the sensor network through long-range communication (such as cables or satellite links), and all communication with users of the sensor network goes through the gateway node.[4]

Since sensors are usually not connected to a fixed infrastructure, they use batteries as their main power supply, and preservation of power is one of the main design considerations of a sensor network [34]. This makes reduction of message traffic between sensors very important.

[4]Relaxations of this requirement, such as communication with the network via UAVs or via an arbitrary node are left for future work.

`SELECT`	{attributes, aggregates}
`FROM`	{Sensordata S}
`WHERE`	{predicate}
`GROUP BY`	{attributes}
`HAVING`	{predicate}
`DURATION`	time interval
`EVERY`	time span e

Figure 3: Query Template

2.2 Sensor Data

A sensor node has one or more sensors attached that are connected to the physical world. Example sensors are temperature sensors, light sensors, or PIR sensors that can measure the occurrence of events (such as the appearance of an object) in their vicinity. Thus each sensor is a separate data source that generates records with several fields such as the id and location of the sensor that generated the reading, a time stamp, the sensor type, and the value of the reading. Records of the same sensor type from different nodes have the same schema, and collectively form a distributed table. The sensor network can thus be considered a large distributed database system consisting of multiple tables of different types of sensors.

Sensor data might contain noise, and it is often possible to obtain more accurate results by *fusing* data from several sensors [12]. Summaries or aggregates of raw sensor data are thus more useful to sensor applications than individual sensor readings [21, 10]. For example, when monitoring the concentration of a dangerous chemical in an area, one possible query is to measure the average value of all sensor readings in that region, and report whenever it is higher than some predefined threshold.

2.3 Queries

We believe that declarative *queries* are the preferred way of interacting with a sensor network. Rather than deploying application-specific procedural code expressed in a Turing-complete programming language, we believe that sensor network applications are naturally data-driven, and thus we can abstract the functionality of a large class of applications into a common interface of expressive queries. In this paper, we consider queries of the simple form shown in Figure 3, and we leave the design of a suitable query language for sensor networks to future work. We also extend the template to support nested queries, where the basic query block shown in Figure 3 can appear within the `WHERE` or `HAVING` clause of another query block.

Our query template has the obvious semantics: The *SELECT* clause specifies attributes and aggregates from sensor records, the *FROM* clause specifies the distributed relation of sensor type, the *WHERE* clause filters sensor records by a predicate, the *GROUP BY*

```
SELECT      AVG(R.concentration)
FROM        ChemicalSensor R
WHERE       R.loc IN region
HAVING      AVG(R.concentration) > T
DURATION    (now,now+3600)
EVERY       10
```

Figure 4: Example Aggregate Query

clause classifies sensor records into different partitions according to some attributes, and the *HAVING* clause eliminates groups by a predicate. Note that it is possible to have join queries by specifying several relations in the *FROM* clause.

One difference between our query template and SQL is that our query template has additional support for long running, periodic queries. Since many sensor applications are interested in monitoring an environment over a longer time-period, *long-running queries* that periodically produce answers about the state of the network are especially important. The *DURATION* clause specifies the life time of a query and the *EVERY* clause determines the rate of query answers: we compute a query answer every e seconds (see Figure 3 [21]). We call the process of computing a query answer a *round*. The focus of this paper is the computation of *aggregate queries*, in which a set of sensor readings is summarized into a single statistic.

Note that our query template has only limited usage for event-oriented applications. For example, to monitor whether the average concentration of a chemical is above a certain threshold, we can use the long-running query shown in Figure 4, but there is a delay of 10 seconds between every recomputation of the average. Event oriented applications are an interesting topic for future research, as the query processing strategies that we propose are optimized for long-running periodic queries, and not event-oriented queries and triggers.

3 Simple Aggregate Query Processing

A *simple aggregate query* is an aggregate query without Group By and Having clauses, a very popular class of queries in sensor networks [21]. In this section we outline how to process such simple aggregate queries. Query processing strategies for more general queries are discussed in Section 5.

3.1 In-Network Aggregation

A query plan for a simple aggregate query can be divided into two components. Since queries require data from spatially distributed sensors, we need to deliver records from a set of distributed nodes to a central destination node for aggregation by setting up suitable communication structures for delivery of sensor records within the network. We call this part of a query plan its *communication component*, and we call the destination node the *leader* of the aggregation. In addition, the query plan has a *computation component* that computes the aggregate at the leader and potentially computes already partial aggregates at intermediate nodes.

Recall that power is one of the main design desiderata when devising query processing strategies for sensor networks. If we coordinating both the computation and communication component of a query plan, we can compute partial aggregates at intermediate nodes as long as they are well-synchronized; this reduces the number of messages sent and thus saves power. We address synchronization in the next section, and consider here three different techniques on how to integrate computation with communication:

Direct delivery. This is the simplest scheme. Each source sensor node sends a data packet consisting of a record towards the leader, and the multi-hop ad-hoc routing protocol will deliver the packet to the leader. Computation will only happen at the leader after all the records have been received.

Packet merging. In wireless communication, it is much more expensive to send multiple smaller packets instead of one larger packet, considering the cost of reserving the channel and the payload of packet headers. Since the size of a sensor record is usually small and many sensor nodes in a small region may send packets simultaneously to process the answer for a round of a query, we can *merge* several records into a larger packet, and only pay the packet overhead once per group of records. For exact query answers with holistic aggregate operators like Median, packet merging is the only way to reduce the number of bytes transmitted [10].

Partial aggregation. For distributive and algebraic aggregate operators [10], we can incrementally maintain the aggregate in constant space, and thus push partial computation of the aggregate from the leader node to intermediate nodes. Each intermediate sensor node will compute partial results that contain sufficient statistics to compute the final result.

3.2 Synchronization

To perform packet merging or partial aggregation, we need to coordinate sensor nodes within the communication component of a query plan. A node n needs to decide whether other nodes $n_1, \ldots, n_k$ are going to route data packets through n; in this case n has the opportunity of either packet merging or partial aggregation. Thus a node n needs to build a list of nodes it is expecting messages from, and it needs to decide how long to wait before sending a message to the next hop.

For duplicate sensitive aggregate operators, like SUM and AVG, one prerequisite to perform partial aggregation is to send each record only once, otherwise duplicate records might appear in partially aggregated

results and bias the result, thus a simple spanning tree might be a suitable communication structure. For other aggregate operators, including MAX and MIN, it is possible to send multiple copies of a record along different paths without any influence on the query accuracy; thus a suitable communication structure might be a DAG rooted at the leader.

The task of *synchronization* in this tree or DAG is then for each node in each round of the query to determine how many sensor readings to wait for and when to perform packet merging or partial aggregation.

Incremental Time Slot Algorithm. Let us first discuss the following simple algorithm. At the beginning of a round, each sensor node sets up a timer, and waits for a special *waiting time* for data packets from its children in the spanning tree or DAG to arrive. The length of the timer at node n is set to the depth of the structure rooted at n times a standardized time slot. However, this algorithm has a large cost in reality. First, it is very difficult to determine in advance how long a node needs to collect records from its children. The time to process the data, schedule the packet, reserve the channel, and retransmit packets due to frequently temporary link failures can vary significantly. Although the expected size of a time slot is small, it has a heavy tail with a big variance. But if the time slot is too large, the accumulated delay at the leader could be very long if the depth of the tree or DAG is large.

Second, with frequent link failures, it is expensive to update the time-out value every time the structure of the communication structure changes. Although most broken links can be repaired locally, repairs may effect the depth of a large number of nodes, and it is expensive to update the timer for all of these nodes. Third, sensor nodes are never completely time-synchronized unless expensive time synchronization protocols or frequent GPS readings are used.

Our Approach. We take a very pragmatic approach to synchronization. Note that for a long-running query, the communication behavior between two sensors n and p is consistent over short periods of time, so it is possible to use historical information to predict future behavior. Assuming that p is the parent of node n. After p receives a record from n, it may expect to receive another record from n in the next round, and thus p adds n to its waiting list. However, such prediction may fail in two cases. First, the parent of node n may change in the next round if n reroutes and has a new parent due to network topology changes and route updates. Second, n could perform a local selection on its records, and only send a record to p if the selection condition is satisfied. Such conditions are only satisfied from time to time, and make the prediction at p fail.

In our approach, we use a timer to recover from false prediction at parent nodes. On the other hand, since a child node is able to determine whether its parent is expecting a packet from it, the child can generate a *notification packet* if its parent's prediction is wrong. We found that this bi-directional prediction approach model the relationship between the parent and child nodes very well in practice, as shown in Section 6.

4 Routing and Crash Recovery

To execute simple aggregate queries, sensor nodes have to send their records to a leader, aggregate them into a final result, and then deliver the final result to the gateway node. Note that a sensor node can only communicate directly with other nodes in its vicinity, limited by the transmission power of the wireless radio. To send messages to a distant node, a multi-hop route connecting the node to the destination has to be established in advance. A packet is forwarded by internal nodes along the route until the packet reaches its destination. Note that this structure is similar in both wired and wireless networks, but there are major differences. In a wired network, the network structure is almost fixed and most routing problems are handled at a few backbone routers. In a wireless network, such as a sensor network, limited connectivity requires all nodes to participate in routing. In addition, the low quality of the communication channel and frequent topology changes make the network quite unstable. Thus more complicated routing protocols are required for wireless networks.

The networking community has developed many different ad-hoc network routing algorithms. A separate *routing layer* in the protocol stack provides a send and receive interface to the upper layer and hides the internals of the wireless routing protocol. In this section, we show that a routing layer for a query processing workload has slightly different requirements than a traditional ad-hoc routing layer, and then we outline some initial thoughts on how to adapt AODV [26], a popular wireless routing protocol, to a query processing workload.

4.1 Wireless Routing Protocols

The two main tasks of a routing protocol are route discovery and route maintenance. Route discovery establishes a route connecting a pair of nodes when required by the upper layer, and route maintenance repairs the route in case of link failures. Many wireless routing protocols have been proposed and implemented, mostly aimed at ad-hoc networks. A distributed and adaptive routing protocol, in which nodes share the routing decision and nodes can change routes according to the network status, is more suitable to sensor networks. Such protocols can be further classified into proactive, reactive and hybrid routing protocols. *Proactive routing protocols*, like DSDV [25], may set up routes between any pair of nodes in advance; while *reactive routing protocols* create and repair routes only

on demand. *Hybrid routing protocols*, e.g. ZRP [11], combine both properties of proactive and reactive protocols.

AODV is a typical reactive routing algorithm. It builds routes between nodes only as desired by the application layer. There are several reasons why we use AODV as the routing protocol for our study. First, reactive routing protocols scale to large-size networks, such as sensor network with thousands of nodes. Second, AODV does not generate duplicate data packets, which is a requirement to do in-network aggregation for duplicate-sensitive aggregate operators. Finally, AODV is a popular ad-hoc network routing protocol and it is implemented in several simulators. Although our discussion is based on AODV, our observations apply to other routing protocols as well.

4.2 Extensions to the Network Interface

Recall from Section 3 that we can optimize aggregate operators through in-network aggregation, such as packet merging and partial aggregation at internal nodes. These techniques require internal nodes to *intercept* data packets passing through them to perform packet merging or partial aggregation. However, with the traditional "send and receive" interfaces of the network layer, only the leader will receive the data packets. The network layer on an internal node will automatically forward the packages to the next hop towards the destination, and the upper layer is not aware of data packets traveling through the node. This functionality is sufficient for direct delivery of packets to a destination node, but to implement in-network aggregation, a node needs the capability to "intercept" packages that are not destined for itself; the query layer needs a way to communicate to the network layer which and when it wants to intercept packages that are destined for the leader [14].

With *filters* [14], the network layer will first pass a package through a set of registered functions that can modify (and possibly even delete) the packet. In case of the query layer, if a node n is scheduled to aggregate data from all children nodes, it can intercept all data packets received from the children nodes and cache the aggregated result. At a specific time, n will generate a new data packet and send it to the leader. All this happens completely transparently to the network layer.

4.3 Modifications to Wireless Routing Protocols

Existing wireless routing protocols are not designed for the communication patterns exhibited by a query processing layer: they are designed for point-to-point communication, and are usually evaluated by selecting two random nodes and establishing and maintaining a communication path between them. A sensor network with a query layer has a significantly different communication pattern: Many source nodes send tuples to a common node, like a leader of an aggregation, or a gateway node. In addition, in a regular ad-hoc network, a node has no knowledge about the communication intents of neighboring nodes, whereas in a sensor network, data transfer to the leader node is usually synchronized to perform aggregation. Thus a node can often estimate when neighboring nodes (such as children in a spanning tree) will send messages to it. We describe here a series of enhancements to one specific routing protocol, AODV [26], although we believe that our techniques are general enough to apply to any wireless routing protocol.

Route initialization. Before sending data packets to the leader, each sensor has to establish a route to the leader, or determine who is the next hop in the DAG or spanning tree. Instead of initializing the route for each node separately from the source node as it would happen in AODV, we can create all the routes together by broadcasting a route initialization message originating at the leader of the aggregation. The message contains a hop count which is used for nodes to determine their depth in the tree. Using this initial broadcast, nodes can save the reverse path as the route to the leader.

Route maintenance. Reliability plays a very important role in in-network aggregation. Since each data packet contains an aggregate result from multiple sensor nodes, dropping a data packet, especially if near the leader, will seriously decrease the accuracy of the final result. The problem is more serious in sensor networks, in which link or node failures happens frequently. We describe two techniques that improve AODV in case of failures.

Local Repair. In AODV, when a broken link is detected, the source node n broadcasts a request to find an alternative route. An internal node n' cannot reply to the request unless n' has a "fresher route" to the leader than n. The efficiency of the local repair algorithm depends on how fast a node can find an up-to-date route in its neighborhood, and AODV uses a sequence number to reflect route "freshness". Given that query processing has a very regular communication structure, in which many of nodes want to route packets to the same destination, we can extend AODV's idea of a sequence number to repair broken routes more efficiently. Since a broken link has no effect on other nodes which are close to the leader, we integrate the depth of a node into the packet sequence number to differentiate sequence numbers between nodes that are spatially close. The new algorithm does not depend on the exact depth of a node to compute the new sequence number; a rough approximation that preserves relative depths is sufficient. Using an approximation to depth prevents a node from updating the depths of all nodes on the path to the leader after the broken

route is repaired, which is a very expensive operation.

Bunch Repair. Local repair can find a new route to bypass a broken link or node in the neighborhood, but it may fail if significant topology changes happen, or a large number of links fail simultaneously due to a spatial disturbance (e.g., large noise in an area). In this case, it is cheaper to repair all routes directly from the leader (by re-broadcasting the route initialization message). Some feedback is required at the leader to active this operation to avoid unnecessary re-initialization. In this first version of our query layer, we re-broadcast a the tree initialization message whenever we receive less than a user-defined fraction of all tuples within an user-defined time interval. (We can calculate the number of tuples that contributed to an aggregate query by adding a COUNT attribute to the partial state of all aggregates.)

5 Query Plans

In this section, we outline the structure of a query plan and discuss general techniques to process sensor network queries.

5.1 Query Plan Structure

Let us consider an example query that we will use to illustrate the components of a query plan. Consider the query "What is the quietest open classroom in Upson Hall?".[5] Assume that the computation plan for this query is to first compute the average acoustic value of each open classroom and then to select the room with the smallest number. There are two levels of aggregation in this plan: (1) to compute the average value of each qualified classroom, and (2) to select the minimum average over all classrooms. The output of the first level aggregation is the input to the second level aggregation.

Users may pose even more complicated queries with more levels of aggregations, and more complex interactions. A query plan decides how much computation is pushed into the network and it specifies the role and responsibility of each sensor node, how to execute the query, and how to coordinate the relevant sensors. A query plan is constructed by *flow blocks*, where each flow block consists of a coordinated collection of data from a set of sensors at the *leader node* of the flow block. The task of a flow block is to collect data from the relevant sensor nodes and to perform some computation at the destination or internal nodes. A flow block is specified by different parameters such as the set of source sensor nodes, a leader selection policy, the routing structure used to connect the nodes to the leader (such as a DAG or tree), and the computation that the block should perform.

[5]Upson Hall is a building with several classrooms located on the Cornell Campus.

A query plan consists of several flow blocks. Creating a flow block and its associated communication and computation structure (which we also call a *cluster*) uses resources in the sensor network. We need to expend messages to maintain the cluster through a periodical heart beat message in which the leader indicates that it is still alive; in case the cluster leader fails, a costly leader election process is required. In addition, a cluster might also induce some delay, as it coordinates computation among the sensors in the cluster. Thus if we need to aggregate sensor data in a region, we should reuse existing clusters instead of creating a new cluster, especially if the data sources are loosely distributed over a larger area, in which case the maintenance cost increases. On the other hand, we should create a flow block if it significantly reduces the data size at the leader node and saves costly transmission of many individual records.

It is the optimizer's responsibility to determine the exact number of flow blocks and the interaction between them. Compared to a traditional optimizer, we would like to emphasize two main differences. First, the optimizer should try to reduce communication cost, while satisfying various user constraints such as accuracy of the query, or a user-imposed maximum delay of receiving the query answers. The second difference lies in the building blocks of the optimizer. Whereas in traditional database systems a building block is an operator in the physical algebra, our basic building block in a sensor database system is a flow block, which specifies both computation and communication within the block.

5.2 Query Optimization

In this section we will discuss how to create a good query plan for more complicated queries. Our discussion stays at the informal level with the goal to help us decide what meta-data we need for the optimizer in the systems catalog. We would like to emphasize that creation of the best query plan for an arbitrary query is a hard problem, and our work should be considered as an initial step towards the design and implementation of a full-fledged query optimizer. We leave experimental evaluations of different query plans to section 6, and the design and implementation of a full-fledged optimizer to future work.

Extension to GROUP BY and HAVING Clauses. Let us consider an aggregate query with `GROUP BY` and `HAVING` clauses. The following query computes the average value for each group of sensors and filters out groups with average smaller than some threshold.

```
(Q1) SELECT     D.gid, AVG(D.value)
     FROM       SensorData D
     GROUP BY   D.gid
     HAVING     AVG(D.value)>Threshold
```

There are two alternative plans for this query. We can create a flow block for each group, or we can create a flow block that is shared by multiple groups. To create a separate flow block can aggregate sensor records of the same group as soon as possible, shorten the path length, and allow to apply the predicate of the HAVING clause to the aggregate results earlier, which saves more communication if the selectivity of the predicate is low. The optimizer should take several parameters into account to make the best plan. One parameter is the overlap of the distribution of the physical locations of the sensors that belong to the different groups. If sensors that below to a single group are physically close, it is better to create a separate flow block to aggregate them together, since the communication cost to aggregate close-by sensors is usually low. However, if sensors from different groups are spatially interspersed, it is more efficient to construct a single flow block shared by all groups.

Joins. The computation part of a flow block does not need to be an aggregate operator. It is possible to add join operators to our query template and define flow blocks with joins. Joins will be common in applications for tracking or object detection. For example, a user may pose a query to select all objects detected in both regions R1 and R2. The following query has a join operator to connect sensor detections in the two regions.

```
(Q2) SELECT   oid
     FROM     SensorData D1, SensorData D2
     WHERE    D1.loc IN R1 AND D2.loc IN R2
              AND D1.oid = D2.oid
```

Join operators represent a wide range of possible data reductions. Depending on the selectivity of the join, it is possible to either reduce or increase the resulting data size. If the join increases the result size, it is more expensive to compute the join result at the leader instead of having the leader send out the tuples from the base relation. Relevant catalog data to make an informed decision concerns the selectivity of the join and the location of the leader.

6 Experimental Evaluation

6.1 Experimental Setup

We have started to implement a prototype of our query processing layer in the ns-2 network simulator [4]. Ns-2 is a discrete event simulator targeted at simulating network protocols to highest fidelity. Due to the strong interaction between the network layer and our proposed query layer, we decided to simulate the network layer to a high degree of precision, including collisions at the MAC layer, and detailed energy models developed by the networking community. In our experiments, we used IEEE 802.11 as the MAC layer [36], setting the communication range of each sensor to $50m$ and assuming bi-directional links; this is the setup used in most other papers on wireless routing protocols and sensor networks in the networking community [14]. In our energy model the receive power dissipation is 395mW, and the transmit power dissipation is 660mW [14]. (This matches numbers from previous studies.) There are many existing power-saving protocols that can turn the radio to idle [35, 27], thus we do not take the energy consumption in the idle state into account. Sensor readings were modeled as 30 bytes tuples.[6]

6.2 Simple Aggregate Query

Let us first investigate experimentally the effects of in-network aggregation. We run a simple aggregate query that computes the average sensor value over all sensor nodes every 10 seconds for 10 continuous rounds. Sensors are randomly distributed in a query region with different size. The gateway node, which is located in the left-upper corner of the query region, is the leader of the aggregate query. Each experiment is the average of ten runs with randomly generated maps.

We first investigate the effect of in-network aggregation on the average dissipated energy per node assuming a fixed density of sensor nodes throughout the network (in this experiment we set the average sensor node density to 8 sensors in a region of $100m \times 100m$).

Figure 5 shows the effect of increasing the number of sensors on the average energy usage of each sensor. In the best case, every sensor only needs to send one merged data packet to the next hop in each round, no matter how many sensors are in the network. The packet merge curve increases slightly as intermediate packets get larger as the number of nodes grows. Without in-network aggregation, a node n has to send a data packet for each node whose route goes through n, so energy consumption increases very fast.

We also investigated the effect of in-network aggregation on the delay of receiving the answer at the gateway node as shown in Figure 6. When the network size is very small, in-network aggregation introduces little extra delay due to synchronization, however as the network size increases, direct delivery induces much larger delay due to frequent conflicts of packets at the MAC layer.

6.3 Routing

To test the efficiency of our improved local repair algorithm, we ran a simple aggregate query which computes the average over all sensor readings every 10 seconds. In this experiment, 200 sensors are randomly distributed in a 500m*500m area. (For other experiments the numbers were qualitatively similar.) We

[6]See the discussion of future work in Section 8 for drawbacks of our current experimental setup.

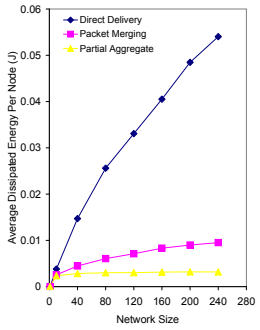

Figure 5: Average Dissipated Energy

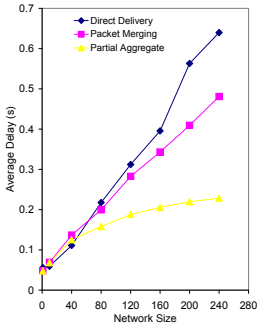

Figure 6: Average Delay vs. Network Size

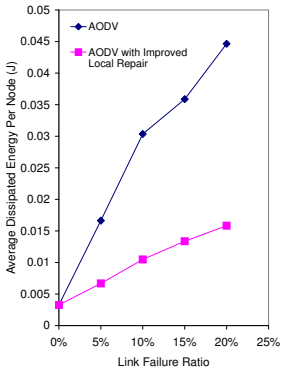

Figure 7: Improved Local Repair Algorithm

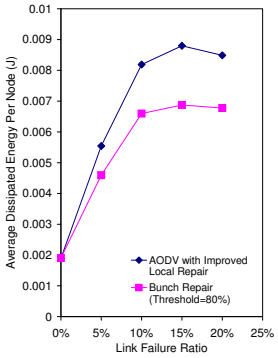

Figure 8: Effect of Bunch Repair

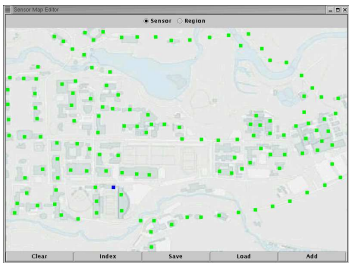

Figure 9: Cornell Map Used in Exp.

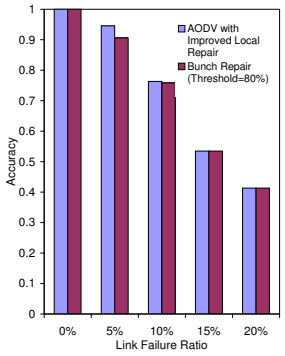

Figure 10: Result Accuracy

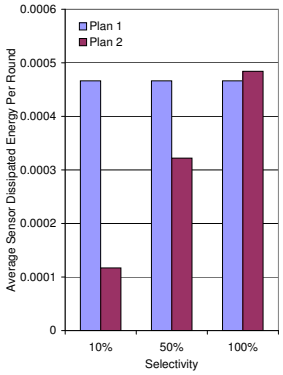

Figure 11: Aggregate Query

introduced random link failures quantified as the percentage of crashed links in a round, and we tested their influence on the routing protocols. Figure 7 shows the comparison between AODV and AODV with our improved local repair algorithm using different link failure rates. As the link failure rate increases, AODV uses much more energy than the algorithm with improved local repair.

We evaluated bunch repair experimentally using the Cornell campus as the query region. About 150 sensors are virtually distributed close to buildings or along main streets; see Figure 9. Figure 8 compares the improved version of AODV with and without bunch repair. The threshold to active route reinitialization is to 80 percent of the tuples. The two algorithms are very close when link failure ratio is low, but our new algorithm saves much energy after the failure ratio becomes larger. This is because bunch repair generates much fewer route request and reply messages, especially when more links fail simultaneously.

Figure 10 shows the influence of bunch repair on the final result accuracy.[7] If several local repairs fail a serious topology change happens, and thus many nodes are disconnected temporarily. A bunch repair will be automatically activated in this case, and thus the accuracy of AODV with bunch repair does not decrease compared to AODV while at the same time the average dissipated energy per node will be much lower compared to AODV.

6.4 Query Plans

We first investigate the benefit of creating a flow block according to the data reduction rate at the leader using the following query:

```
(Q3) SELECT    AVG(value)
     FROM      Sensor D
     WHERE     D.loc IN [(400,400),(500,500)]
     HAVING    AVG(value)>t
```

[7] The accuracy is measured at the end of each round. Packets are dropped at internal nodes if they belong to the previous round.

Let us consider two different query plans for Query Q3. In Plan 1, we use an existing flow block which covers the whole network. This flow block is also used to collect system catalog information, thus it does not incur additional maintenance cost. In Plan 2, we construct a new flow block for Query Q3 just inside the query region, where we first compute the average at a leader of this block, and then send qualifying averages to the gateway node.

Both plans may perform the `HAVING` operation as a filter over the average value of the sensor reading, at the gateway for Plan 1, but the leader node in Plan 2. The result in Figure 11 shows that if the selectivity of the `HAVING` operator is close to 100% and thus the computation at the leader does not reduce the number of outgoing averages, then there is not much difference in terms of the average dissipated sensor energy for each round of the query. Plan 2 spends only a little more energy on maintenance of the additional flow block. However, as the leader discards aggregated data packets with higher and higher probability, Plan 2 is a much better choice. It reduces the traffic flow significantly through aggregation at the leader much closer to data sources, compared to the gateway node of Plan 1.

Next, we evaluate different query plans for Query Q1, a query with a GROUP BY clause. Assume that there are four different groups. Let us consider three simple cases: In the *distributed* case, sensors that below to a single group are physically close, but far away from other groups. In the *close-by* case, the groups are close to each other, but they do not overlap, whereas in the *overlap* scenario, all four groups are in the same area. We again consider two different query plans for this query. Plan 1 creates one big cluster to be shared by all groups. Aggregation within each group happens at the global cluster leader. Plan 2 creates a separate cluster for each group, and aggregates only the tuples relevant for each group at the respective cluster leader. Figure 16 shows the different spatial distributions of the four groups for the three cases.

We can see from Figure 12 that if the groups are physically close, then there is no big difference between Plans 1 and 2. However, creating one big cluster increases the connectivity of the cluster, and reduces the risk of network partitioning within a cluster. If the four groups are spatially distant from each other, Plan 2 is more efficient as the selection at the aggregation leaders can reduce the number of data packets for transmission back to the gateway node. In the last scenario, where the different groups of sensors are randomly distributed, Plan 1 outperforms Plan 2, since the cost to collect data records at the leader is high.

Figures 13 and 14 show the influence of operator selectivity on the two plans in the previous experiment for two different sensor topologies, the distributed topology in Figure 13 and the overlap topology in Figure 14. The experiment shows that operator selectivity has a strong influence on plan performance, although the sensor topology has a much larger impact.

Next we consider the Join Query Q2. Again we consider two query plans to evaluate this query. In Plan 1, sensors send all tuples back to the gateway without any in-network computation; Plan 2 creates a flow block for the Join operator inside the query region. In Plan 2, in case the join reduces the data size at the leader, the leader sends the result of the join back to the gateway, otherwise, the leader sends all individual data records to the gateway for the join to be performed there.

Figure 15 shows that the cost to collect data at the leader is non-trivial. If the join operator at the leader fails to reduce the data size, then the total energy consumption at the node increases. Thus the optimizer needs to estimate the selectivity of the join operator, and it needs statistics in the systems catalog to make the right decision.

7 Related Work

Research of routing in ad-hoc wireless networks has a long history [17, 30], and a plethora of papers has been published on routing protocols for ad-hoc mobile wireless networks [25, 16, 5, 26, 24, 8, 15]. All these routing protocols are general routing protocols and do not take specific application workloads into account, although we believe that most of these protocols can be augmented with the techniques similar to those that we propose in Section 4. The SCADDS project at USC and ISI explores scalable coordination architectures for sensor networks [9], and their data-centric routing algorithm called directed diffusion [14] first introduced the notion of filters that we advocate in Section 4.2.

There has been a lot of work on query processing in distributed database systems [40, 7, 23, 39, 18], but as discussed in Section 1, there are major differences between sensor networks and traditional distributed database systems. Most related is work on distributed aggregation, but existing approaches do not consider the physical limitations of sensor networks [33, 38]. Aggregate operators are classified by their properties by Gray et al. [10], and an extended classification with properties relevant to sensor network aggregation has been proposed by Madden et al. [21].

The TinyDB Project at Berkeley also investigates query processing techniques for sensor networks including an implementation of the system on the Berkeley motes and aggregation queries [19, 20, 21, 22].

Other relevant areas include work on sequence query processing [31, 32], and temporal and spatial databases [41].

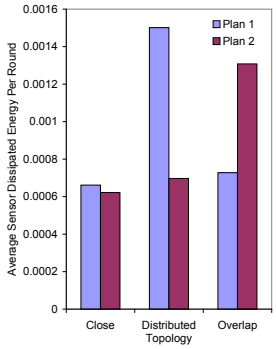

Figure 12: Impact of Sensor Distributions to Different Query Plan

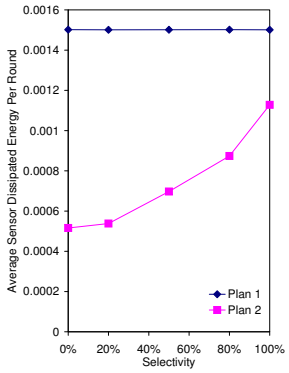

Figure 13: Distributed Topology

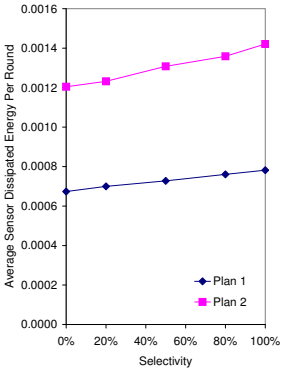

Figure 14: Overlap Topology

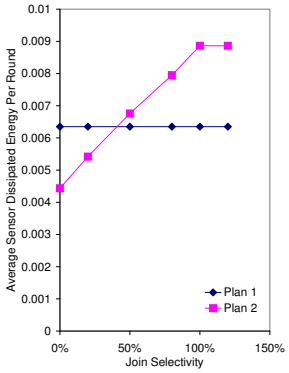

Figure 15: Join Query

Physical distribution:	Location of the four sensor groups
`distributed`	[100,100,200,200] [100,400,200,500] [400,100,500,200] [400,400, 500,500]
`close-by`	[100,100,200,200] [100,200,200,300] [200,100,300,200] [200,200,300,300]
`overlap`	Sensors of all groups are randomly distributed [100,100,300,300]

Figure 16: Query Characteristics

8 Conclusions

Sensor networks will become ubiquitous, and the database community has the right expertise to address the challenging problems of tasking the network and managing the data in the network. We described a vision of processing queries over sensor networks, and we discussed some initial steps in in-network aggregation, implications on the routing layer, and query optimization. We have started at Cornell to design and implement a prototype that allows us to experiment with the design space of various algorithms and data structures [6].

Future work. This work opens a plethora of new research directions at the boundary of database systems and networking. First, we believe that TDMA MAC protocols will be very important in power-constrained sensor networks [27], and we plan to investigate the interaction of a TDMA MAC layer with routing and query processing in future work. In addition, our current simulation assumes bidirectional links, which is usually not true in practice. Having filters as an additional interface to the routing layer leaves many open questions, such as an efficient implementation of filters, the order in which filters should be evaluated, handling of conflicting actions, etc. We assumed very simple SQL blocks as query templates without discussing a full-fledges spatio-temporal query language whose design is a challenging topic for future work. In addition, we only scratched the surface of query processing, metadata management, and query optimization, and much work needs to be done multi-query optimization, distributed triggers, and the design of benchmarks. We anticipate that the emergence of new applications, as well as the implementation and usage of our prototype system will lead to other research directions. We believe that sensor networks will be a fruitful research area for the database community for years to come.

Acknowledgments. We thank the DARPA SensIT community for helpful discussions. Praveen Sheshadri and Philippe Bonnet made influential initial contributions to Cougar. The Cornell Cougar Project is supported by DARPA under contract F-30602-99-0528, NSF CAREER grant 0133481, the Cornell Information Assurance Institute, Lockheed Martin, and by gifts from Intel and Microsoft. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] www.microsoft.com/windows/embedded/ce.net.

[2] www.redhat.com/embedded.

[3] ACM SIGMOBILE. *Proceedings of MOBICOM 1998*. ACM Press.

[4] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu. Advances in network simulation. *IEEE Computer*, 33(5):59–67, May 2000.

[5] J. Broch, D. A. Maltz, D. B. Johnson, Y.-C. Hu, and J. Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. [3], pages 85–97.

[6] M. Calimlim, W. F. Fung, J. Gehrke, D. Sun, and Y. Yao. Cougar Project web page. www.cs.cornell.edu/database/cougar.

[7] S. Ceri and G. Pelagatti. *Distributed Database Design: Principles and Systems*. MacGraw-Hill (New York NY), 1984.

[8] S. Das, C. Perkins, and E. Royer. Performance comparison of two on-demand routing protocols for ad hoc networks. In *INFOCOM 2000*, pages 3–12. IEEE.

[9] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next century challenges: Scalable coordination in sensor networks. In *MOBICOM 1999*, pages 263–270. ACM Press.

[10] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[11] Z. Haas. The zone routing protocol (ZRP) for wireless networks. IETF MANET, Internet Draft, 1997.

[12] D. L. Hall and J. Llinas, editors. *Handbook of Multisensor Data Fusion*. CRC Press, 2001.

[13] J. Hill and D. Culler. A wireless embedded sensor architecture for system-level optimization. Submitted for publication, 2002.

[14] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *MOBICOM 2000*, pages 56–67. ACM Press.

[15] P. Johansson, T. Larsson, N. Hedman, B. Mielczarek, and M. Degermark. Scenario-based performance analysis of routing protocols for mobile ad-hoc networks. In *MOBICOM 1999*, pages 195–206. ACM Press.

[16] D. B. Johnson and D. A. Maltz. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*. Kluwer Academic Publishers, 1996.

[17] J. Jubin and J. D. Tornow. The DARPA packet radio network protocol. *Proceedings of the IEEE*, 75(1):21–32, Jan. 1987.

[18] D. Kossmann. The state of the art in distributed query processing. *Computing Surveys*, 32, 2000.

[19] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE 2002*.

[20] S. Madden and J. M. Hellerstein. Distributing queries over low-power wireless sensor networks. In *SIGMOD 2002*.

[21] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. In *OSDI 2002*.

[22] S. R. Madden, R. Szewczyk, M. J. Franklin, and D. Culler. Supporting aggregate queries over ad-hoc sensor networks. In *Workshop on Mobile Computing and Systems Applications (WMCSA)*, 2002.

[23] M. T. Özsy and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, Englewood Cliffs, 1991.

[24] V. D. Park and M. S. Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *INFOCOM 1997*. IEEE.

[25] C. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *SIGCOMM 1994* , pages 234–244. ACM Press.

[26] C. E. Perkins. Ad hoc on demand distance vector (aodv) routing. Internet Draft, http://www.ietf.org/internet-drafts/draft-ietf-manet-aodv-04.txt, October 1999.

[27] G. J. Pottie and W. J. Kaiser. Embedding the Internet: wireless integrated network sensors. *Communications of the ACM*, 43(5):51–51, May 2000.

[28] G. J. Pottie and W. J. Kaiser. Wireless integrated network sensors. *Communications of the ACM*, 43(5):51–58, 2000.

[29] V. Raghunathan, C. Schurgers, S. Park, and M. B. Srivastava. Energy-aware wireless microsensor networks. *IEEE Signal Processing Magazine*, 19(2):40–50, 2002.

[30] N. Schacham and J. Westcott. Future directions in packet radio architectures and protocols. *Proceedings of the IEEE*, 75(1):83–99, January 1987.

[31] P. Seshadri, M. Livny, and R. Ramakrishnan. Seq: A model for sequence databases. In *ICDE 1995*. IEEE Computer Society.

[32] P. Seshadri, M. Livny, and R. Ramakrishnan. The design and implementation of a sequence database system. In *VLDB 1996*, pages 99–110. Morgan Kaufmann.

[33] A. Shatdal and J. F. Naughton. Adaptive parallel aggregation algorithms. In *SIGMOD 1995*, pages 104–114.

[34] T. Simunic, H. Vikalo, P. Glynn, and G. D. Micheli. Energy efficient design of portable wireless systems. In *ISLPED 2000*, pages 49–54. ACM Press.

[35] S. Singh, M. Woo, and C. S. Raghavendra. Power-aware routing in mobile ad hoc networks. [3], pages 181–190.

[36] I. C. Society. Wireless LAN medium access control (mac) and physical layer specification. IEEE Std 802.11, 1999.

[37] M. Stonebraker. Operating system support for database management. *CACM*, 24(7):412–418, 1981.

[38] W. P. Yan and P.-Å. Larson. Eager aggregation and lazy aggregation. In *VLDB 1995*. Morgan Kaufmann.

[39] C. Yu and W. Meng. *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann, San Francisco, 1998.

[40] C. T. Yu and C. C. Chang. Distributed query processing. *ACM Computing Surveys*, 16(4):399–433, Dec. 1984.

[41] C. Zaniolo, C. S., C. Faloutsos, R. Snodgrass, V. S. Subrahmanian, and R. Zicari, editors. *Advanced Database Systems*. Morgan Kaufmann, San Francisco, 1997.

Appendix E:

COUGAR: The Network is the Database

Wal Fu Fung
Cornell University
326 Upson Hall
Ithaca, NY 14853, USA
waifu@cs.cornell.edu

David Sun
Cornell University
326 Upson Hall
Ithaca, NY 14853, USA
davidsun@cs.carnell.edu

Johannes Gehrke
Cornell University
41 05B Upson Hall
Ithaca, NY 14853, USA
johannes@cs.cornell.edu

1. INTRODUCTION

The widespread distribution and availability of small-scale sensors, actuators, and embedded processors is transforming the physical world into a computing platform. One such example is a sensor network consisting of a large number of sensor nodes that combine physical sensing capabilities such as temperature, light, or seismic sensors with networking and computation capabilities (1]. Applications range from environmental control, warehouse inventory, health care to military environments. Existing sensor networks assume that the sensors are preprogrammed and send data to a central frontend where the data is aggregated and stored for offline querying and analysis. This approach has two major drawbacks. First, the user cannot change the behavior of the system on the fly. Second, communication in today's networks is orders of magnitude more expensive than local computation, thus in-network processing can vastly reduce resource usage and thus extend the lifetime of a sensor network.

This demo demonstrates a database approach to unite the seemingly conflicting requirements of scalability and flexibility in monitoring the physical world. We demonstrate the COUGAR System, a new distributed data management infrastructure that scales with the growth of sensor inter-connectivity and computational power on the sensors over the next decades. Our system resides directly on the sensor nodes and creates the abstraction of a single processing node without centralizing data or computation.

2. THE COUGAR SYSTEM

The COUGAR System is a platform for testing query processing techniques over ad-hoc sensor networks. COUGAR has a three-tier architecture: The *QueryProxy,* a small database component that runs on sensor nodes to interpret and execute queries, and a *Frontend* component, which is a more powerful *QueryProxy* that permits connections to the world outside of the sensor network, and a graphical user interface through which users can pose ad-hoc and long-running queries on the sensor network. Our system forms clusters out of the sensors to allows intelligent in-network aggregation to conserve energy by reducing the amount of communication between sensor nodes. The query processing component handles queries for distributed devices in an intelligent manner.

ACM SIGMOD '2002 June 4-6, Madison, Wisconsin, USA

3. THE DEMO

This demonstration will utilize Sensoria WINSNG 2.0 nodes [4], running Linux on SH4 CPUs, as well as Berkeley Motes [2]. Each of the Sensoria nodes has GPS, seismic, and acoustic sensors, while each Mote has light and temperature sensors. The Sensoria nodes will run the *QueryProxy* software, while the Motes will be running a scaled down version of the *QueryProxy.* The sensor network will consist of Sensona nodes and Motes communicating via RF radio using the Directed Diffusion routing protocol [3] with an XML message format. The GUI will communicate with a Sensoria node running the front-end over Ethernet.

The demonstration will illustrate the *QueryProxy* system's ability to interact with different sensor types and hardware, dynamically obtain available sensor types via an in-network catalog, and aggregate query responses in-network. Users will be able to create and execute their own queries over the sensor network.

4. ACKNOWLEDGEMENTS

The COUGAR Project is supported by the Defense Advanced Research Project Agency, the Cornell Information Assurance Institute, and by a gift from Intel.

5. REFERENCES

[1] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next century challenges: Scalable coordination in sensor networks. pages 263—270.

[2] J. Hill, R. Szewczyk, A. Woo, D. Culler, S. Hollar, and K. Pister. System architecture directions for networked sensors. *ACM SIGPLAN Notices,* 35(11):93-104, Nov. 2000.

[3] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. pages 56—67.

[4] 5. C. www.sensoria.com. Wins ng 2.0 user guide. White paper, July 2001.